CIF documentation

Copyright (c) 2010, 2026 Contributors to the Eclipse Foundation

Version v13.0-RC1

Table of Contents

1. Synthesis-based engineering . . . 3
   1.1. Supervisory controllers . . . 3
   1.2. Engineering approaches for supervisory controller development . . . 5
      1.2.1. Traditional engineering . . . 7
      1.2.2. Model-based engineering . . . 10
      1.2.3. Verification-based engineering . . . 15
      1.2.4. Synthesis-based engineering . . . 16
   1.3. Synthesis-based engineering example . . . 23
      1.3.1. FIFO requirement . . . 24
      1.3.2. Synthesis-based engineering . . . 25
      1.3.3. Example benefits of synthesis-based engineering . . . 26
   1.4. Synthesis-based engineering in practice . . . 27
      1.4.1. Development process . . . 29
      1.4.2. Advanced topics . . . 36
      1.4.3. Synthesis-based engineering in practice example . . . 39
   1.5. Challenges in applying synthesis-based engineering . . . 46
      1.5.1. Change in way-of-working . . . 47
      1.5.2. Tool support . . . 48
2. Language tutorial . . . 49
   2.1. Introduction . . . 49
   2.2. Lessons . . . 49
   2.3. Basics . . . 56
      2.3.1. Automata . . . 56
      2.3.2. Synchronizing events . . . 58
      2.3.3. Non-determinism . . . 61
      2.3.4. Alphabet . . . 62
      2.3.5. Event declaration placement . . . 64
      2.3.6. Shorter notations . . . 66
   2.4. Data . . . 68
      2.4.1. Discrete variables . . . 69
      2.4.2. Discrete variable value changes . . . 69
      2.4.3. Location/variable duality (1/2) . . . 70
      2.4.4. Location/variable duality (2/2) . . . 71
      2.4.5. Global read, local write . . . 73
      2.4.6. Monitoring . . . 74
      2.4.7. Old and new values in assignments . . . 78
      2.4.8. The `tau` event . . . 81
      2.4.9. Initial values of discrete variables . . . 83

2.4.10. Initialization predicates . . . 85
2.4.11. Using locations as variables . . . 89
2.4.12. State (exclusion) invariants . . . 91
2.4.13. State/event exclusion invariants . . . 94
2.5. Types and values . . . 98
2.5.1. Types, values, and expressions . . . 98
2.5.2. Values overview . . . 98
2.5.3. Integers . . . 99
2.5.4. Ranged integers . . . 100
2.5.5. Reals . . . 100
2.5.6. Booleans . . . 102
2.5.7. Strings . . . 102
2.5.8. Enumerations . . . 102
2.5.9. Tuples . . . 103
2.5.10. Lists . . . 105
2.5.11. Bounded lists and arrays . . . 108
2.5.12. Sets . . . 109
2.5.13. Dictionaries . . . 110
2.5.14. Combining values . . . 112
2.5.15. If and switch expressions . . . 113
2.6. Scalable solutions and reuse (1/2) . . . 115
2.6.1. Constants . . . 115
2.6.2. Algebraic variables . . . 116
2.6.3. Algebraic variables and equations . . . 118
2.6.4. Type declarations . . . 120
2.7. Time . . . 121
2.7.1. Timing . . . 121
2.7.2. Continuous variables . . . 122
2.7.3. Continuous variables and equations . . . 124
2.7.4. Equations . . . 126
2.7.5. Variables overview . . . 127
2.7.6. Urgency . . . 128
2.7.7. Deadlock and livelock . . . 129
2.8. Channel communication . . . 130
2.8.1. Channels . . . 130
2.8.2. Dataless channels . . . 134
2.8.3. Combining channel communication with event synchronization . . . 135
2.9. Functions . . . 137
2.9.1. Functions . . . 137
2.9.2. Internal user-defined functions . . . 137
2.9.3. Function statements . . . 139

2.9.4. Functions as values . . . 143
2.10. Scalable solutions and reuse (2/2) . . . 144
2.10.1. Automaton definition/instantiation . . . 144
2.10.2. Parametrized automaton definitions . . . 146
2.10.3. Automaton definition parameters . . . 147
2.10.4. Groups . . . 154
2.10.5. Group definitions . . . 155
2.10.6. Imports . . . 159
2.10.7. Imports and libraries . . . 160
2.10.8. Imports and groups . . . 161
2.10.9. Namespaces . . . 164
2.10.10. Input variables . . . 165
2.11. Stochastics . . . 168
2.11.1. Stochastics . . . 168
2.11.2. Discrete, continuous, and constant distributions . . . 169
2.11.3. Pseudo-randomness . . . 172
2.12. SVG visualization and interaction . . . 173
2.12.1. SVG visualization . . . 173
2.12.2. First example . . . 175
2.12.3. Inkscape . . . 176
2.12.4. Sun/moon example . . . 179
2.12.5. Walk example . . . 180
2.12.6. Rate example . . . 184
2.12.7. Workstation example . . . 186
2.12.8. Tank example . . . 189
2.12.9. Lamps example . . . 192
2.12.10. Buffers/products example . . . 196
2.12.11. SVG interaction . . . 198
2.13. Text . . . 201
2.13.1. Print output . . . 201
2.13.2. Print output examples . . . 202
2.13.3. Text formatting . . . 208
2.14. Synthesis . . . 212
2.14.1. Supervisory controller synthesis . . . 212
2.14.2. Automaton kinds . . . 213
2.14.3. Invariant kinds . . . 214
2.14.4. Event controllability . . . 215
2.14.5. Marker predicates . . . 216
2.15. Annotations . . . 217
2.15.1. Annotations . . . 217
2.15.2. Controller properties annotations . . . 218

2.15.3. Documentation annotations . . . 218
2.15.4. Partial specification annotations . . . 219
2.15.5. Reachability requirement annotations . . . 220
2.15.6. State annotations . . . 225
3. Language reference . . . 227
3.1. Syntax . . . 228
3.1.1. Lexical syntax . . . 228
3.1.2. Grammar . . . 233
3.1.3. CIF XML files . . . 247
3.2. SVG visualization and interaction . . . 247
3.2.1. CIF/SVG declarations . . . 248
3.3. Text . . . 266
3.3.1. Print declarations . . . 266
3.3.2. Print file declarations . . . 272
3.3.3. Text formatting details . . . 275
3.4. Annotations . . . 286
3.4.1. Annotations . . . 286
3.4.2. Built-in annotations . . . 291
4. Tools . . . 298
4.1. Specification tools . . . 299
4.1.1. CIF text editor . . . 299
4.2. Analysis tools . . . 300
4.2.1. CIF information generator . . . 300
4.2.2. CIF relations collector . . . 302
4.3. Supervisory controller synthesis tools . . . 306
4.3.1. Data-based supervisory controller synthesis . . . 306
4.3.2. CIF multi-level splitter . . . 360
4.3.3. Event-based synthesis toolset . . . 367
4.3.4. CIF to Supremica transformer . . . 386
4.4. Simulation, validation, and verification tools . . . 392
4.4.1. CIF simulator . . . 392
4.4.2. CIF to mCRL2 transformer . . . 480
4.4.3. CIF to UPPAAL transformer . . . 488
4.4.4. Controller properties checker . . . 491
4.5. Real-time testing, code generation, and implementation tools . . . 514
4.5.1. CIF code generator . . . 514
4.5.2. CIF PLC code generator (stable) . . . 556
4.5.3. CIF PLC code generator (deprecated) . . . 582
4.6. Miscellaneous tools . . . 600
4.6.1. CIF to CIF transformer . . . 600
4.6.2. CIF merger . . . 716

4.6.3. CIF event disabler . . . 725
4.6.4. CIF explorer . . . 728
4.6.5. CIF to yEd transformer . . . 731
4.7. Scripting . . . 740
4.7.1. Introduction to scripting . . . 740
4.7.2. Overview of scriptable tools . . . 744
5. CIF examples . . . 748
6. CIF release notes . . . 749
6.1. Version 13.0 . . . 749
6.2. Version 12.0 (2026-06-30) . . . 750
6.3. Version 11.0 (2026-03-31) . . . 751
6.4. Version 10.0 (2025-12-19) . . . 755
6.5. Version 9.0 (2025-09-30) . . . 758
6.6. Version 8.0 (2025-06-30) . . . 761
6.7. Version 7.0 (2025-03-31) . . . 765
6.8. Version 6.0 (2024-12-20) . . . 768
6.9. Version 5.0 (2024-10-03) . . . 772
6.10. Version 4.0 (2024-06-30) . . . 778
6.11. Version 3.0 (2024-03-31) . . . 784
6.12. Version 2.0 (2023-12-22) . . . 787
6.13. Version 1.0 (2023-09-30) . . . 789
6.14. Version 0.10 (2023-06-30) . . . 790
6.15. Version 0.9 (2023-03-31) . . . 793
6.16. Version 0.8 (2022-12-21) . . . 795
6.17. Version 0.7 (2022-09-30) . . . 796
6.18. Version 0.6 (2022-07-07) . . . 797
6.19. Version 0.5 (2022-03-29) . . . 798
6.20. Version 0.4 (2021-12-17) . . . 799
6.21. Version 0.3 (2021-10-01) . . . 801
6.22. Version 0.2 (2021-07-07) . . . 802
6.23. Version 0.1 (2021-04-02) . . . 804
7. Developers . . . 805
7.1. Extracting information from a CIF specification . . . 805
7.1.1. Recognizing text of a CIF specification . . . 805
7.1.2. Structure of the CIF object-tree . . . 806
7.1.3. Getting relevant information from the object-tree . . . 809
7.1.4. Advanced access or modification of a CIF specification . . . 811
7.1.5. The CIF meta-model . . . 812
7.2. CIF language modification . . . 813
8. CIF history . . . 816
8.1. Conception . . . 816

8.2. CIF 1 . . . 817
8.3. CIF 2 . . . 817
8.4. CIF 3 . . . 818
8.5. Eclipse ESCET . . . 819
9. Legal . . . 821
Index . . . 823

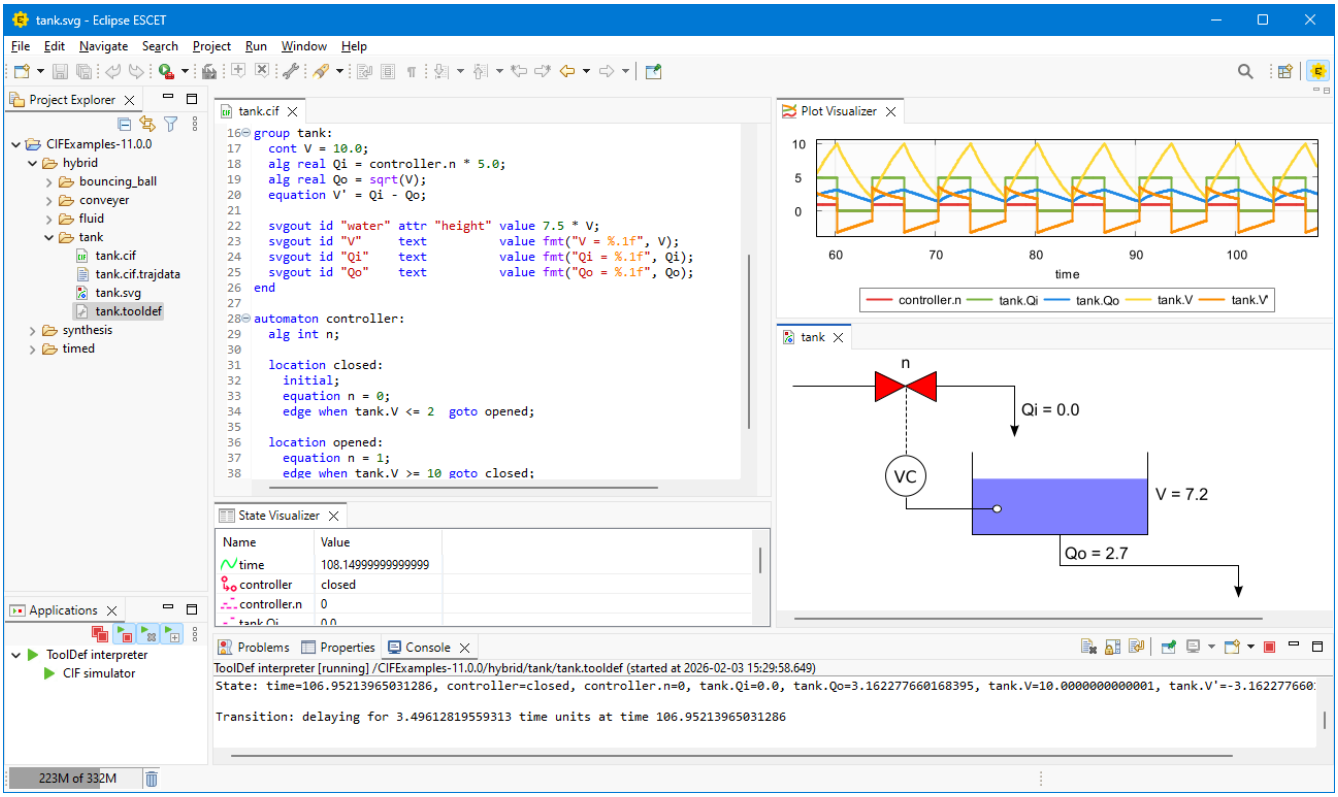
CIF is a declarative modeling language for the specification of discrete event, timed, and hybrid systems as a collection of synchronizing automata. The CIF tooling supports the entire development process of controllers, including among others specification, supervisory controller synthesis, simulation-based validation and visualization, verification, real-time testing, and code generation. Combined they enable a synthesis-based engineering approach to efficiently and cost-effectively design and implement high-quality controllers.

CIF is one of the tools of the Eclipse ESCET™ project. Visit the project website for downloads, installation instructions, source code, general tool usage information, information on how to contribute, and more.

The documentation consists of:

- CIF synthesis-based engineering manual
- CIF language tutorial
- CIF language reference manual
- CIF tool manual
- CIF examples
- CIF release notes
- CIF developers manual
- CIF history (and why it's called 'CIF')
- Legal information

A screenshot showing a CIF model and simulation:


tank.svg - Eclipse ESCET
File Edit Navigate Search Project Run Window Help
Project Explorer
CIFExamples-11.0.0
hybrid
bouncing_ball
conveyer
fluid
tank
tank.cif
tank.cif.trajdata
tank.svg
tank.tooldef
synthesis
timed
tank.cif
16 group tank:
17 cont V = 10.0;
18 alg real Qi = controller.n * 5.0;
19 alg real Qo = sqrt(V);
20 equation V' = Qi - Qo;
21
22 svgout id "water" attr "height" value 7.5 * V;
23 svgout id "V" text value fmt("V = %.1f", V);
24 svgout id "Qi" text value fmt("Qi = %.1f", Qi);
25 svgout id "Qo" text value fmt("Qo = %.1f", Qo);
26 end
27
28 automaton controller:
29 alg int n;
30
31 location closed:
32 initial;
33 equation n = 0;
34 edge when tank.V <= 2 goto opened;
35
36 location opened:
37 equation n = 1;
38 edge when tank.V >= 10 goto closed;
State Visualizer
Name
Value
time
108.14999999999999
controller
closed
controller.n
0
Plot Visualizer
10
5
0
60
70
80
90
100
time
controller.n
tank.Qi
tank.Qo
tank.V
tank.V'
tank
n
Qi = 0.0
VC
V = 7.2
Qo = 2.7
Applications
ToolDef interpreter
CIF simulator
Problems
Properties
Console
ToolDef interpreter [running] /CIFExamples-11.0.0/hybrid/tank/tank.tooldef (started at 2026-02-03 15:29:58.649)
State: time=106.95213965031286, controller=closed, controller.n=0, tank.Qi=0.0, tank.Qo=3.162277660168395, tank.V=10.0000000000001, tank.V'=-3.162277660
Transition: delaying for 3.49612819559313 time units at time 106.95213965031286
223M of 332M

Chapter 1. Synthesis-based engineering

CIF supports synthesis-based engineering (SBE), an engineering approach to design and implement supervisory controllers. The engineering approach combines model-based engineering with computer-aided design to produce correct-by-construction controllers. It does so by automating as many steps as possible in the development process.

Supervisory controller synthesis is a main element of synthesis-based engineering, and a key feature of CIF. It involves the automatic generation of supervisory controller models. Implementation of the controller is achieved through (implementation language) code generation, improving speed and reducing the number of errors introduced at this stage. Combined, they allow engineers to focus on *what* the controller should do, rather than *how* it should do it, and how this is to be implemented.

Synthesis-based engineering has many more benefits. As the engineering approach uses well-defined models, designs can be discussed, analyzed, model-checked, or simulated. That allows finding and correcting issues early in the development process, rather than during later stages where correcting them is more costly. It also supports a comprehensive modular design and efficient incremental engineering. Ultimately, this reduces development time and improves the quality of the resulting supervisory controllers.

Supervisory controllers

Explains what supervisory controllers are, in what types of systems you can find them, and where they are located within such systems.

Engineering approaches for supervisory controller development

Discusses synthesis-based engineering of supervisory controllers, its benefits, and its relation to other engineering approaches.

Synthesis-based engineering example

Demonstrates the value of synthesis-based engineering through an example.

Synthesis-based engineering in practice

Explains concretely how to use the CIF language and toolset to apply synthesis-based engineering of supervisory controllers.

Challenges in applying synthesis-based engineering

Explains the challenges of embedding a synthesis-based engineering approach into industrial practice.

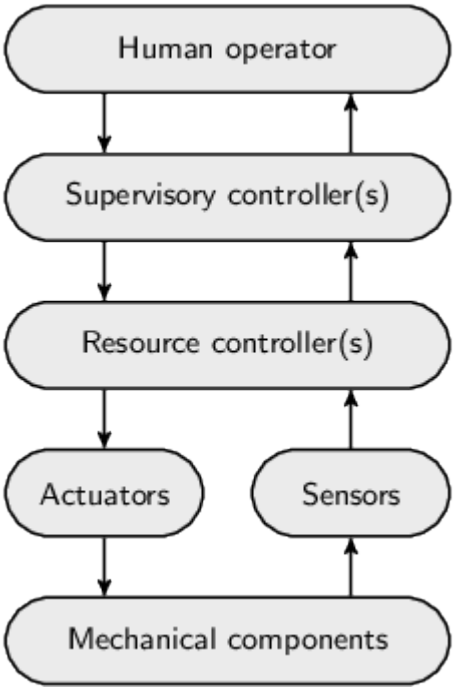
To learn how to use SBE, you can also follow the Eclipse ESCET online SBE course.

1.1. Supervisory controllers

Automated systems are all around us. For instance, hospitals use X-ray and MRI systems, industrial printers print books and magazines, lithography systems are essential for the production of computer chips, and waterway locks bring ships from one water level to another. In today's digital age, all these systems contain software that controls their operation. Such systems are often called cyber-physical systems, for the *physical* part that consists of the hardware components being controlled, and the *cyber* part that contains the software that controls those physical parts.

Today's cyber-physical systems are often highly complex. To manage their complexity, they are typically step-by-step divided into sub-systems, sub-sub-systems, etc, each with their own responsibility. This way, at the most subdivided level, *components* are obtained that are small enough to be developed, tested and maintained in isolation. The components are often divided over several layers, to form a layered system architecture.

The control of a system can similarly be subdivided and layered. The following figure shows a traditional view on the control of a system:



At the bottom are the mechanical components, such as motors, switches, levers and valves. Their operation can be steered through actuators and their state of operation can be observed through sensors. Resource controllers provide a first level of control. They may for instance correct for sensor jitter, translate continuous signals to discrete ones, or detect and even correct anomalous situations.

A supervisory controller provides higher-level control. It is typically responsible for the correct and safe behavior of a (sub-)system. For instance, it could be responsible for preventing damage to mechanical components or human operators. It may prevent collisions with or among mechanical components, or prevent mechanical components from overheating. It could control a single sub-system, coordinating one or more resource controllers. However, in case of a layered architecture, it could also coordinate multiple supervisory controllers of a lower layer. Supervisory controllers

can thus be found at various levels of a system architecture.

Some systems are fully automated, without the need for human intervention or control. However, most systems provide some kind of human-machine interface that allows a human operator to monitor the system and if necessary control its operation.

Regardless of the exact system architecture, and whether human intervention is possible or not, supervisory controllers play an essential role in the safe control of all kinds of cyber-physical systems, and can be found at various levels within such systems.

1.2. Engineering approaches for supervisory controller development

Supervisory controllers can be developed in various ways. The following figure gives an overview. It shows multiple approaches to design and engineer supervisory controllers, as well as how these approaches relate to each other:

Engineering approach → / ↓ Development step	Traditional Engineering	Model-Based Engineering	Verification-Based Engineering	Synthesis-Based Engineering
Requirements design	Document-based	Document-based	**Model-based (formal)**	Model-based (formal)
Controller design	Document-based	**Model-based (formal)**	Model-based (formal)	**Computer-aided (formal)**
Realization in software (implementation code)	Traditional software engineering (coding)	**Code generation (fault-free code)**	Code generation (fault-free code)	Code generation (fault-free code)
Verification (against requirements)	Testing	Testing + **Model-based testing**	**Formal verification (model checking)**	**Correct-by-construction (guaranteed)**
Validation (of requirements)	Testing	Testing + **Simulation**	Testing + Simulation	Testing + Simulation

Legend: Manual work / Focus (Semi-)automatic

The columns indicate various engineering approaches. From left to right, they employ progressively more automation and computer assistance. The rows of the table indicate typical steps involved in the development of supervisory controllers. The cells indicate for each approach what is involved in the particular step. The green-colored cells indicate that the step involves mostly, or at least significant, manual work for the particular engineering approach. Contrarily, the gray-colored cells indicate that the step is (mostly) automated for that approach. As the steps with more manual work generally require the most engineering effort, the green cells also indicate where the engineering focus is for a particular approach. Bold texts in cells indicate changes compared to the previous column.

Typical steps involved in the development of supervisory controllers, as represented by the rows from top to bottom, are:

- **Requirements design** focuses on what a controller for a (sub-)system must do. Functional and safety requirements may be specified, for instance requiring that pushing an emergency button stops all motors. Extra-functional requirements may also be specified, for instance requiring that a certain throughput should be achieved.

- **Controller design** focuses on how a controller should satisfy the requirements to efficiently and safely control the system. For instance, the various control states of the system may be specified, as well as how the controller reacts to changing sensor or other input signals by controlling actuators, e.g., enabling a motor.

- Controllers may be **realized in software**. The software source code may for instance be implemented using Java, C or PLC programming languages.

- **Verification** involves checking the realized controller against its requirements design and controller design, to ensure that the controller is correctly realized. The system, controlled by the controller, should behave as designed.

- **Validation** involves checking the realized controller and its design, to ensure that the right controller is made. That is, the requirements must be correct and complete, such that the controller ensures that the system operates safely and efficiently in all circumstances.

Typically, the various engineering approaches as represented by the columns from left to right, can be characterized as follows:

- **Traditional engineering** is document-based. Requirements are written down informally in large requirement documents. They are used as input for controller design documents. The documents are then handed over to a different person, team or supplier, for the implementation. Implementation of the controller in software is done through manual coding. Verification and validation involve testing at various levels, including unit testing, integration testing and system testing. Traditionally, all five steps are performed manually, which is laborious and error-prone.

- **Model-based engineering** or **model-driven engineering** automates the realization step, and provides computer assistance especially for the verification and validation steps. It places models at the center of attention. The controller is modeled in a formal way, allowing a computer to interpret and analyze its behavior. That is, it is specified in a mathematically unambiguous way, for instance using state machines. Such models are considered the single source of truth. From them, all kinds of artifacts can be generated automatically, including the software code of the controller's implementation. This ensures that the code is fault-free and behaves consistently with the behavior as expressed by the controller model. The models can also be used to partly automate verification, for instance through the use of model-based testing. Simulation models can be used to simulate the (controlled) system behavior and validate the requirements during early phases of development.

- **Verification-based engineering** is a form of model-based engineering with computer-assistance to automate the verification step. It uses formal verification, or model checking, a mathematical technique that can automatically check the controller model against its requirements. To employ formal verification, both the controller model and the requirements must be formally specified. Formal verification then either indicates that the specified requirements are guaranteed to be satisfied by the controller model, or it provides counter examples that indicate in which situations they are not satisfied. This is exhaustive, as it considers every conceivable scenario, unlike testing, which typically covers only a limited number of scenarios. Through formal verification, the controller model can be iteratively adapted to satisfy all specified requirements in every possible situation.

- **Synthesis-based engineering** is a form of model-based engineering with computer-assistance to automate the design of the controller. It uses supervisory controller synthesis to automatically synthesize a controller model from requirements and a simple model of the to-be-controlled system. This mathematical technique guarantees that the synthesized controller model satisfies all specified requirements. This makes verification of the controller model against the specified requirements superfluous, as the synthesized controller model is guaranteed correct-by-construction. With the controller design, realization and verification either to a large degree being automated or unnecessary, the focus shifts to requirements design and validation. This allows engineers to focus on *what* the controller should do, rather than *how* it should achieve it.

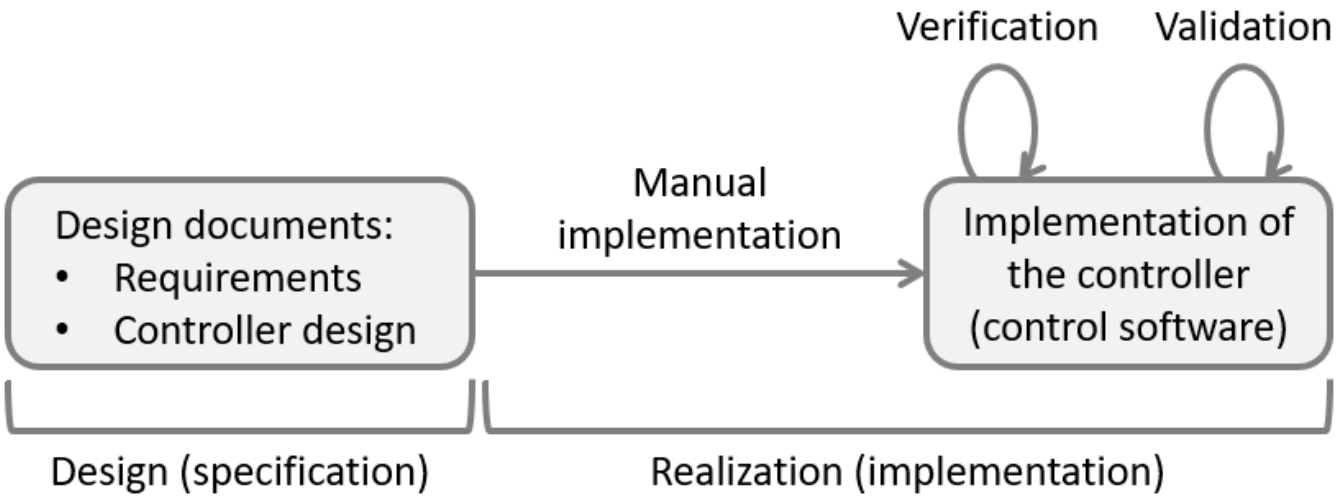
The use of model-based engineering combined with computer-aided design, through formal methods like formal verification and supervisory controller synthesis, has many advantages. It allows to produce unambiguous, complete, consistent, and up-to-date specifications, leading to higher quality controllers at similar or even lower effort and costs.

After this general comparison of the approaches, the following provides more detailed information:

- Traditional engineering
- Model-based engineering
- Verification-based engineering
- Synthesis-based engineering

1.2.1. Traditional engineering

The following figure shows a simplified development process for traditional engineering of supervisory controllers:



Traditionally, controllers are first specified in design documents. They for instance list their functional and safety requirements, describe their control states and indicate when the controller should actuate the various actuators depending on changing sensor signals.

Subsequently, the controller is manually implemented in software code through the use of a programming language, such as PLC code for a PLC platform, or Java or C++ code for an industrial PC.

Finally, the implementation is verified and validated, typically by means of testing. Verification involves checking and ultimately ensuring that the controller satisfies its specified requirements. Validation involves checking that the controller exhibits the desired behavior, and thus ensuring it is the desired controller. Since a controller must satisfy its specified requirements, this includes validating the requirements to ensure they are the desired requirements.

Downsides of traditional engineering

Traditional engineering has been around for a long time. Companies typically know what works and what doesn't, and how to work around the various challenging aspects of it. It can work well, particularly for small and simple systems, developed by a well-managed but small team. However, the approach has several disadvantages. These become especially apparent when applying it to develop controllers for larger and more complex systems, developed by multiple teams, or with some development activities outsourced to suppliers:

Ambiguity

It is extremely difficult to unambiguously write down the control requirements in a document. Often textual descriptions in natural languages can be interpreted in various ways.

The domain expert who writes the requirements has a certain mental picture in their mind. However, software engineers responsible for realizing these requirements in the software implementation may interpret them differently after forming their own mental picture. There is often a big gap between the specification of the design and its implementation.

The documents may also serve as input or as a contract to a supplier to develop the control

software. Then the impact and costs of ambiguity can be huge, much more so than when the implementation is done in-house within the company.

Incompleteness and inconsistency

Besides the interpretation of the requirements also their completeness and consistency is important. Often the normally occurring situations (happy flow) is adequately covered by the requirements. However, the edge cases and exceptional circumstances are just as important, especially when safety is of critical importance to the system.

Consider for instance requirements for when the hardware fails, such as when a cable breaks or a sensor becomes defect. Such cases are often far more complex and the number of combinations/interactions that has to be considered can be immense. Ensuring that the textual descriptions of all these cases do not lead to inconsistencies is often practically undoable.

A good domain expert will be able to limit the number of mistakes, such as missing requirements and contradictions in the requirements specification, but typically can't completely eliminate them. A good software/PLC engineer will surely spot some of the remaining mistakes during the implementation and testing of the controller.

However, even thoroughly tested and delivered industrial code often still contains faults. Furthermore, if the specification is incomplete, software engineers will make their own choices, which may or may not match with what the domain expert had in mind. Again, working with external suppliers, rather than doing the development in-house within the company, may aggravate these concerns.

Multi-disciplinary systems

The multi-disciplinary nature of design versus implementation also plays a role. A domain expert may know everything about the functional requirements of the system. The software engineer, especially one from a supplier, may lack such knowledge. They come from different domains, often use different technical terms, and thus essentially speak different languages. This makes it more difficult for them to understand each other, and hinders communication.

Abstraction levels

Furthermore, there is a difference in level of abstraction between design and realization. The control requirements are often written as functional specifications. For the implementation numerous details of a lower abstraction level play a role, such as data structures, message encodings and byte orderings. A functional specification typically does not concern itself with such aspects. Again, people from different disciplines and domains may not be able to effectively communicate with each other.

Mixing design with implementation aspects

The situation becomes even more complex if (unintentionally) during the design also implementation aspects are incorporated into the functional specification. Then the clear separation between design and realization is lost. This often leads to more misunderstandings, which then requires more communication and collaboration to resolve.

Outdated documentation

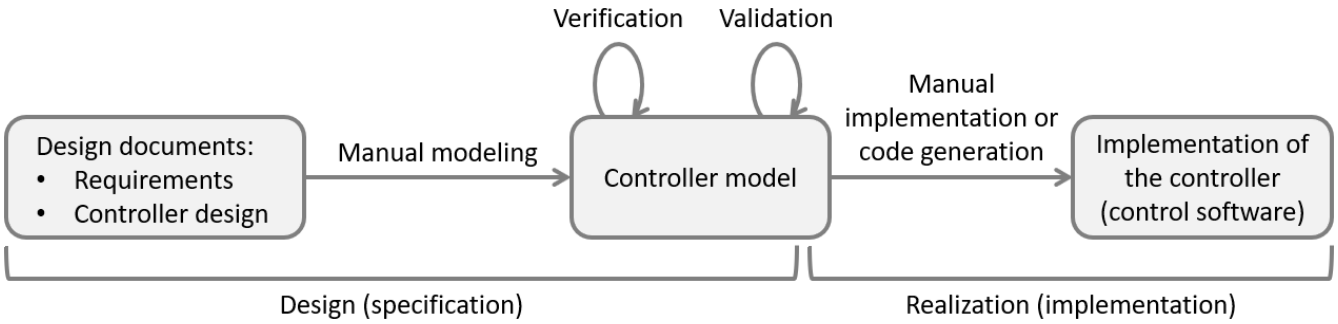
Another aspect to consider for specifications in documentation, is that any changes, such as bug fixes and new features, are often only implemented in the software. After a while the documents become more and more outdated and thus unusable. This increases the gap between specification and implementation.

1.2.2. Model-based engineering

Model-based design, model-based software/system engineering and model-driven engineering, are related terms. They place models at the center of the entire development process and the entire lifecycle of the system, including design, implementation and maintenance. The models fill the gap between the specification and implementation.

Model-based engineering process

The following figure shows a simplified development process for model-based engineering of supervisory controllers:



At the center is a controller model, a model of a controller that unambiguously specifies how the controller works. It precisely specifies how the state of the controller changes when a sensor signal changes, and under what conditions and in which states an actuator may be turned on or off. Ideally, the model has a mathematical foundation. It may for instance be modeled as one or more state machines.

The controller model is manually modeled from design documents. They for instance list the functional and safety requirements of the controller, describe its control states and indicate when it should actuate the various actuators depending on changing sensor signals.

The controller model must be verified and validated. Verification involves checking and ultimately ensuring that the system, controlled by the controller, satisfies its specified requirements. Validation involves checking that the controller ensures the desired system behavior, and thus ensuring it is the desired controller. Since a controller must satisfy its specified requirements, this includes validating the requirements to ensure they are the desired requirements. This may be

supported by formal methods, methods with a mathematical foundation, and supported by computer tools. For instance, a controller model may be simulated. This may reveal issues, that can be addressed to improve the controller model.

The control software is typically implemented using a programming language, such as PLC code for a PLC platform, or Java or C++ code for an industrial PC. This may for instance be done in-house within the company, by different teams or departments, or by a supplier. While manual implementation is possible, the code is often automatically generated from the controller model.

Benefits of model-based engineering

Model-based engineering directly addresses many of the downsides of traditional engineering:

Unambiguous and intuitive specifications

It is important that the models are formal models, with a mathematical meaning (semantics). Examples of formal models are state machines to model controllers and logical formulas for model requirements. The use of such formal models leads to unambiguous interpretation of control requirements and controller behavior.

The use of the right formal language, in which control requirements can be specified in an intuitive manner is essential. This is where domain specific languages (DLSs) play a role. Such a language closely matches the world of the domain experts, such that they can directly write their control requirements in a notation that fits how they think about the system. This leads to readable and unambiguous specifications.

Besides specific to a domain, domain specific languages are also more restrictive in what you can write down than a general programming language. While this seems to be a limitation, it is actually their strength. Due to the limited number of concepts to consider, there are less different ways to model a system. This further reduces ambiguity, due to more consistency and simpler specifications.

Bridges the multi-disciplinary specification/implementation gap

Using a good domain specific language, both domain experts and software engineers can understand and interpret the specification in the same way, regardless of their different backgrounds. Obviously, the language must be rich enough to properly describe all relevant aspects of the domain. It must also use a proper abstraction level.

Complete and consistent specifications through computer-aided validation and verification

The use of unambiguous formal models has even more advantages, as it makes it possible for a computer to interpret and analyze the models. The limited concepts of the domain specific

language help to do so efficiently and scalably. Computers can with formal methods, mathematical techniques, quickly and accurately analyze countless scenarios. This is a great advantage compared to traditional document reviews.

An example of this is verification by means of model-based testing. Instead of manually writing dozens or hundreds of tests, a computer can automatically generate thousands, millions or even more tests from the controller model. This allows covering much more behavioral scenarios, increasing confidence in correctness of the controller model and its implementation.

Another example of this is validation of the specification by means of simulation. Using simulation various execution scenarios can be examined, to give insight into the behavior of the system being controlled by the controller. This provides new insights that can be used to further improve the specification. Especially for complex situations, which are difficult to understand, this is of great value.

The use of computer-aided verification and validation often exposes issues in the specification. Model-based testing for instance, may find that a certain scenario was not considered during controller design, and therefore does not satisfy the requirements. The controller model may then be adapted and tested again. This allows to effectively and iteratively improve the design, leading to more complete and consistent specifications, and therefore to better quality controllers.

Address issues early to reduce effort and costs

A great benefit of model-based engineering is that verification and validation can be done already during the earlier phases of development, rather than only at later phases such as implementation or testing. It is well-known in industry that the later a mistake is found and fixed, the higher the effort and costs to do so. In practice, implementations developed using model-based engineering approaches are often produced more efficiently and with less mistakes. Through automation, changes can be incorporated more quickly into the models, and these can automatically be analyzed again.

Furthermore, the benefit of discussions that may arise early on during the development process, for instance about how the specification must be adapted if it is found lacking, is not to be underestimated. It is of great value that so early on it is possible to discuss control requirements and the behavior of the system during unforeseen circumstances, such as when a sensor is defect.

Efficiently obtain correct-by-construction implementations

After several iterations the confidence in the controller specification is sufficiently high, and thus the chance of incompleteness and inconsistencies sufficiently low, given the amount of effort and money that can reasonably be spent during the development process. The development process produces an implementation-independent model of the control logic, that during the realization can be implemented. This may be done by a different team or department

within the same company, or even by an external supplier. The formal specifications can then serve as a contract with the third party, allowing for more control. They can also be used to perform acceptance tests on the implementation.

While the controller can be manually implemented based on the controller model, automatic generation of the control software is often a better choice. Automation prevents the kinds of subtle mistakes that humans make when they manually implement something, ensuring consistency between the specification and the implementation. Automation also improves efficiency. If the controller model is changed, with the push of a button a new correct-by-construction implementation can quickly be generated from it.

Implementation-independent models separate design from implementation

Since a controller model is implementation-independent, there is a clear separation between design (specification) and realization (implementation). It allows generating implementation code for different platforms, such as industrial PCs or PLCs, with different programming languages, such as Java, C or PLC code, for 32 or 64 bits architectures, etc. Additionally, controller models are vendor-independent, allowing to for instance generate PLC code for PLCs from different vendors. It is also possible to switch to a different platform or vendor at a later time, or additionally generate code for other platforms or vendors.

Up-to-date models are the single source of truth

Model-based engineering places models at the center of attention. It is the models that are adapted if they are functionally incorrect, have inconsistencies, or new functionality is required. Techniques such as model-based testing, simulation, and code generation all operate on the models. The models are therefore the 'single source of truth'. Contrary to documents, the models will be maintained. They remain up-to-date as they are the basis of all development during the entire life cycle of the system, including design, realization and maintenance.

The use of model-based engineering combined with computer-aided design through formal methods thus has many advantages. It allows for producing unambiguous, complete, consistent, and up-to-date specifications, leading to higher quality controllers at similar or even lower effort and costs. However, specific forms of model-based engineering, such as verification-based and synthesis-based engineering, can offer additional benefits.

Even though model-based engineering has many benefits, companies should not underestimate how significantly different it is from traditional engineering. They should consider and manage the challenges particular to this engineering approach.

Terminology

The following terminology is often used when discussing model-based engineering of supervisory controllers:

Code generation

The automatic generation of correct-by-construction control software from a controller model.

Control requirements

Properties that a system must satisfy, even if they are not satisfied in the uncontrolled system. Examples include functional and safety properties. They are called control requirements, or simply requirements.

Control software

The implementation of the controller in software. For instance, PLC code for a PLC platform, or Java or C++ code for an industrial PC.

Controller model

A model of a controller that unambiguously specifies how the entire controller works. Also called a supervisory controller, or simply controller, in control theory. It precisely specifies how the state of the controller changes when a sensor signal changes, and under what conditions and in which states an actuator may be turned on or off.

Controller validation

The process of checking and ultimately ensuring that the system being controlled by a controller exhibits the desired behavior, and thus ensuring that the controller is the desired controller. Since a controller (model) must satisfy its specified requirements, this includes validating the requirements to ensure they are the desired requirements.

Controller verification

The process of checking and ultimately ensuring that the controller satisfies its specified requirements.

Domain-specific language

A modeling language with concepts specific to a certain domain. This can for be the domain of supervisory controllers with concepts such as plants and requirements, or the domain of office lighting systems with concepts such as lamps and occupancy sensors.

Formal method

A method with a mathematical foundation, typically supported by computer tools. For instance,

formal verification or supervisor synthesis.

Model

An unambiguous representation of all relevant concepts, ideally with a mathematical foundation. For instance, a model of control requirements in the form of logical formulas, or a model of a controller represented as a state machine.

Model-based development/engineering

Places models at the center of the entire development process and the entire lifecycle of the system, including design, implementation and maintenance.

Modeling language

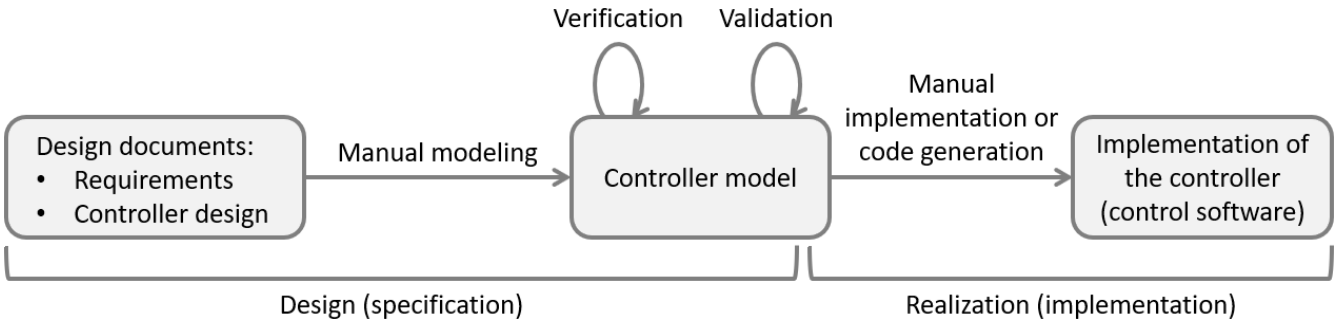
A language in which models can be specified, in an unambiguous way, and ideally also with mathematical foundation.

1.2.3. Verification-based engineering

Verification-based engineering is a form of model-based engineering. It uses formal verification to automate the verification that the controller model satisfies its requirements.

Verification-based engineering process

The following figure shows a simplified development process for verification-based engineering of supervisory controllers:



The verification-based engineering process is very much similar to the model-based engineering process. The only difference is the way the verification of the controller (model) against its specified requirements is performed. Verification-based engineering uses formal verification, or model checking, to mathematically prove a certain property holds. Such properties could for instance be the absence of deadlock or livelock, or that a bridge may only open if its corresponding traffic lights

have been set to signal a red light. Formal verification can prove that such properties hold for every conceivable scenario.

If a property does not hold, formal verification produces counter examples, typically in the form of a sequences of inputs that lead to states in the controller model where the property is not satisfied. This makes it possible to pinpoint the problem in the model, and address it. It is often an iterative process to address such issues, perform verification again, address more issues, perform verification again, etc. If no counter examples are produced, all verified properties are guaranteed to be satisfied by the controller model.

To employ formal verification not only the controller model must be formally specified, but also the properties to check. This means that the requirements are no longer specified in natural language in documents, but in mathematically unambiguous specifications. An example is state machines that define the order in which things may happen, such as that a certain sensor must go on before an actuator can be enabled. Another example is logical formulas that indicate that certain combinations of states in the controller models should never occur, e.g., they could indicate a collision that is to be prevented.

Benefits of verification-based engineering

Verification-based engineering has all the benefits of model-based engineering. Additionally, it has the following benefit:

Formal verification guarantees that the requirements are satisfied

Formal verification considers every conceivable scenario. It can therefore mathematically prove that a specified requirement is satisfied by the controller (model). It is thus more powerful than testing, which typically covers only a limited number of scenarios and is then not exhaustive.

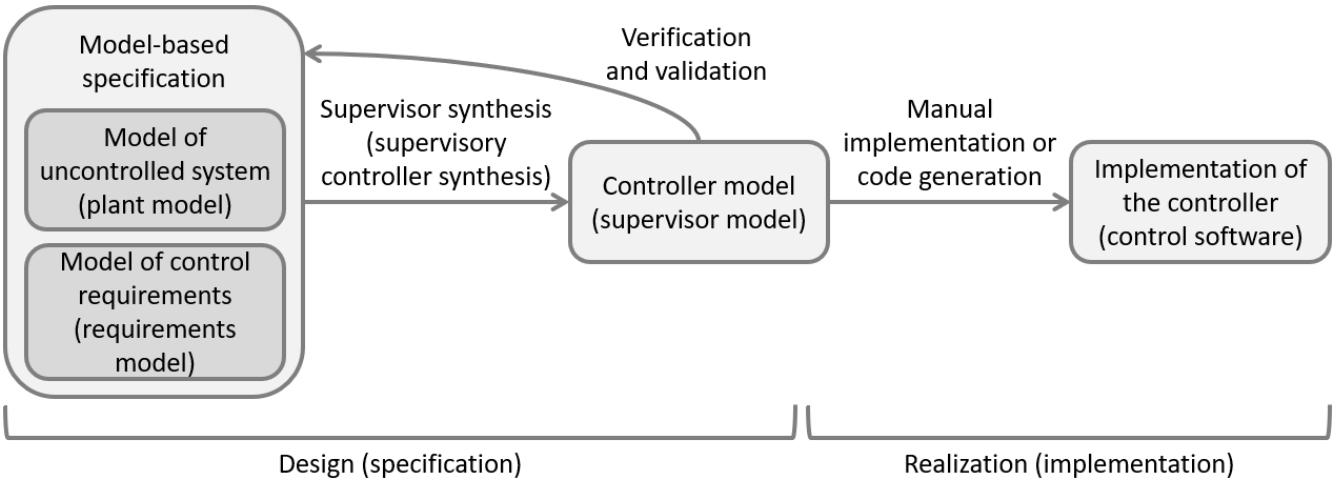
Even though verification-based engineering has many benefits, companies should not underestimate how significantly different it is from traditional engineering or even from lesser-automated forms of model-based engineering. They should consider and manage the challenges particular to this engineering approach.

1.2.4. Synthesis-based engineering

Synthesis-based engineering is a form of model-based engineering. It uses supervisory controller synthesis (or simply supervisor synthesis) to automatically synthesize a correct-by-construction controller model.

Synthesis-based engineering process

The following figure shows a simplified development process for synthesis-based engineering of supervisory controllers:



As with general model-based engineering, at the center is a controller model with a mathematical foundation. From the controller model, the control software can still be manually implemented or automatically generated.

However, with synthesis-based engineering, the controller model is not manually modeled from design documents. Instead, it is automatically generated from models of the uncontrolled system (plant model) and control requirements (requirements model).

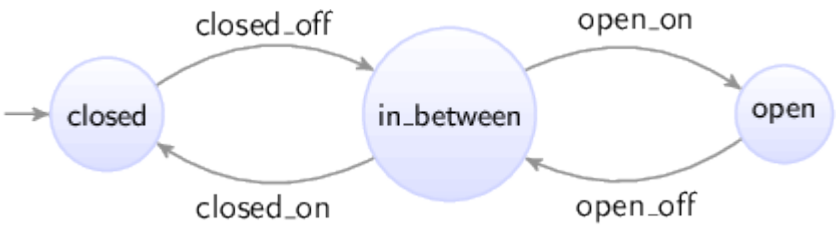
Verification to ensure that the controller (model) satisfies the requirements used for synthesis is then superfluous, as the synthesized controller model is correct-by-construction. Verification may still be needed for additional requirements that are not yet supported by synthesis, such as stronger liveness requirements and timed requirements.

Validation that the controller model behaves as intended is still needed, because even though synthesis guarantees that the synthesized controller correctly adheres to the specified requirements, the specified requirements may not be the desired requirements. For instance, the requirements could be incomplete, too restrictive, or contain other mistakes, resulting in the system being controlled by the controller exhibiting behavior that is not desired or not showing behavior that is desired.

Input: plant and requirements models

Supervisor synthesis requires two types of models as input. The first type of model is called a *plant* model, after the concept of plant from control theory. Plant models describe capabilities or behavior of a physical system 'as is', without any integrated control. They represent the possible behavior of the uncontrolled system. The second type of model is called a *requirements* model. Requirements models describe the requirements that the controller must adhere to. They model restrictions upon the behavior of the plants, to ensure that only the desired behavior remains.

A plant model can for instance specify which sensors and actuators are present in the system. It may also specify their interdependencies. For instance, a sensor that indicates that a gate is open and a sensor that indicates it is closed, can under normal circumstances not be enabled at the same time. A plant model is often modeled as a state machine. The following figure shows as an example a the combined plant model for the two gate sensors:



Initially it is closed. As the gate is opened, the gate closed sensor goes off, and the gate is somewhere in between. Then it can be closed again, making the gate closed sensor go on. But the gate can also keep going further open, until it is fully opened, and the gate open sensor goes on. There is can be closed again, making the gate open sensor go off. The plant model clearly states that it is not possible for the gate to be open and closed at the same time, as the plant model can only be in one state at a time.

Actions, such a sensor going on or off, or an actuator being turned on or off, are called *events*. Supervisor synthesis distinguishes two types of events, *controllable* and *uncontrollable* events. Controllable events can be controlled by the supervisory controller. Actuators are typically modelled as controllable events, such that the supervisory controller decides when to actuate them. Uncontrollable events operate autonomously, from the perspective of the supervisory controller. A controller can not prevent such events from occurring in the system. For instance, a user may push a button and the corresponding sensor will indicate whether the button is pushed or not. The events to indicate changes in the status of the sensor will happen. The supervisor can not prevent this. Another example of uncontrollable events is limit sensors of movements. When a movement is completed, its limit sensor will be activated, leading to an uncontrollable event being fired.

Plant models at the relatively low abstraction level of sensors and actuators are quite common. However, modeling and controlling (sub-)systems at a higher abstraction level is possible as well. See for more information the section on supervisory controllers.

A requirements model captures requirements. It may specify functional requirements, safety requirements, etc. For instance, the motor to open a gate may only be activated once the barrier to stop traffic is fully closed. Control requirements can also be specified as state machines, but often the use of a logical formula is more intuitive. Combining them is also possible. Well-formulated logical formulas are easy to understand, even for people without a mathematical background. As an example, consider a requirement in three forms: natural language, mathematical formula, and modeled in CIF:

- Natural language: "The actuator to open the gate may only be activated if the barrier is fully closed."
- Mathematical formula: gate_open_actuator.c_on ⇒ barrier_sensors.closed
- Modeled in CIF as a state/event exclusion requirement:

```
requirement gate_open_actuator.c_on needs barrier_sensors.closed;
```

Supervisory controller synthesis

Supervisory controller synthesis generates from the plant and requirements models a controller model, a model of the control logic, named a *supervisor* or supervisory controller. The synthesized supervisor is correct-by-construction, i.e., it satisfy all the requirements in every situation.

The supervisor may be represented as another state machine, but it may for instance also be a list of conditions under which actuators may be activated or deactivated. The synthesized supervisor as a state machine, or the plant model together with the supervisor in the form of extra synthesized control conditions, forms the *controlled system*.

The controlled system is guaranteed to satisfy the following properties:

- **Safe:** It satisfies all specified requirements in all situations that the specified uncontrolled system can be in.
- **Controllable:** It only limits controllable events of the plant, e.g., it may prevent enabling or disabling an actuator, but can't prevent a sensor from going on or off.
- **Non-blocking:** It doesn't block, i.e., a marked state can always be reached from every reachable state, thus ensuring a form of liveness.
- **Maximally permissive:** It does not impose more restrictions than strictly necessary to enforce the previous properties, i.e., it is maximally permissive. In other words, the controlled system permits all safe, controllable, and non-blocking behaviors, i.e., it is minimally restrictive.

In practice the terms *supervisor* and *controller* are often used interchangeably. Formally however, they can be different. A *supervisor* is maximally permissive and may still allow multiple (safe) choices, for instance between enabling multiple different actuators, or between enabling one actuator and disabling another one. A *controller* is considered to explicitly choose specific controllable events rather than allowing multiple ones.

Benefits of synthesis-based engineering

Synthesis-based engineering has all the benefits of model-based engineering. Additionally, it has the following benefits:

Computer-aided design for improved quality at reduced effort and cost

Computer-aided design and automation shorten the development cycle and reduce human errors. This improves the quality and reliability of controllers, and reduces effort and costs.

More concretely, supervisor synthesis provides computer-aided design assistance. It can for instance automatically detect conflicting requirements. It will also detect that a certain activator may never be enabled in a certain state, because under certain specific conditions this may *later* lead to an unavoidable unsafe state. For complex systems, this kind of situations are often difficult to foresee for human beings. It is therefore difficult to correctly manually model them in a controller model.

Focus on the *what* rather than the *how*

With synthesis-based engineering the controller model is automatically synthesized. From it, the implementation is automatically produced through code generation. Verification is (to a large degree) not needed as the implementation is correct-by-construction. The focus therefore shifts to requirements design and validation. Engineers can thus focus on '*what* should the controller do' (its requirements), rather than on '*how* must the controller achieve this' (the controller design and implementation).

An example is specifying a First-In-First-Out (FIFO) requirement. While specifying it may be quite easy, realizing the requirement in a controller model may be complex due to the various situations that may arise in the system. Supervisory controller synthesis can evaluate all possible combinations of conditions and synthesize a controller that is mathematically correct for all of them. This kind of design automation is even more useful when multiple, complex and related requirements need to be considered. The synthesized supervisor is correct-by-construction for all requirements in all situations, preventing human errors.

Verification exposes problems, synthesis solves them

Synthesis-based engineering goes far beyond verification-based engineering. Formal verification exposes problems. It tells you that the controller model is not correct and in which situations, and you need to iteratively adapt it yourself. Each time formal verification produces a counter example to indicate a requirement violation, the controller model needs to be manually adapted. Contrary, supervisor synthesis provides solutions. It automatically synthesizes a controller model that satisfies all the requirements. Synthesis produces in one go a supervisor with all the additional conditions that must be enforced to ensure all requirements are satisfied. This makes verification of the controller model against the requirements from which the supervisor was synthesized superfluous, as the synthesized controller model is already correct-by-construction.

Maintain maximum design space freedom

Maximal permissiveness ensures that maximum design space freedom is maintained. When manually designing a controller, an engineer may favor simple control conditions that severely limit the design space. As synthesis produces maximally-permissive supervisors, it imposes minimal restrictions, while still satisfying all requirements. This leaves design space freedom to e.g. choose performance-optimal solution among safe alternatives. A performance-optimal *controller* may for instance be derived from a *supervisor* model that allows multiple (safe) choices.

Supports a comprehensive modular design and efficient incremental engineering

Each part of the plant and each requirement can be specified separately. This way it is easy to adapt specific plants or requirements, or add new ones. Modular specifications thus allow for efficient incremental engineering, as after each change a simple re-synthesis is enough to obtain a new correct-by-construction controller.

Supports reuse and standardization

The separately specified plants and requirements can even be put in libraries with reusable standardized building blocks. This allows engineers to easily build up new specifications from existing proven building blocks, combining them in different ways. Ultimately this leads to more uniformity and improves efficiency.

Intuitive specifications with fine-grained requirement traceability

Each plant and requirement can be specified separately, and has a clear purpose. This provides a good overview of the control requirements, and allows for fine-grained requirement traceability. This unlike the controller model itself. There, one requirement can have an effect on various parts (states) of the controller. It can thus be spread out over the controller model, and mixed with other requirements. Clear modular specifications avoid hiding undesired and unneeded behavior in a large/complex controller.

See the synthesis-based engineering example section for a concrete example that shows the power of synthesis.

Even though synthesis-based engineering has many benefits, companies should not underestimate how significantly different it is from traditional engineering or even from lesser-automated forms of model-based engineering, such as verification-based engineering. They should consider and manage the challenges particular to this engineering approach.

Terminology

The following terminology is often used when discussing synthesis-based engineering of supervisory controllers, in additional to model-based engineering terminology:

Controllable event

An event that is controlled (enabled or disabled) by the controller. Events to actuate (turn on or off) an actuator are often controllable events.

Controlled system

The *uncontrolled system* together with a *supervisor* or *controller that controls it. This may be represented as a single state machine, or as a combination of the _plant* model with the *supervisor* or *controller* model.

Controller

A controller model that explicitly chooses specific controllable events, rather than allowing multiple ones as a *supervisor* may do. When this distinction is not relevant, *supervisor* and *controller* are often used interchangeably.

Correct-by-construction formal method

A formal method that guarantees that the result of the method satisfies all requirements.

Event

An action representing something that can happen in the system. For instance, there may be low-level actions for sensors going on or off, and actuators being turned on or off. There may also be higher level actions, such as a command to move an object from one location to another, a command to turn an entire subsystem on or off, or an event through which a subsystem indicates that an error has occurred.

Synthesis-based engineering

A form of model-based engineering that uses supervisory controller synthesis (or simply supervisor synthesis) to automatically synthesize a correct-by-construction controller model.

Supervisor

A maximally permissive controller model that may still allow multiple (safe) choices. Unlike a *controller*, it may for instance allow a choice between enabling multiple different actuators, or between enabling one actuator and disabling another one. When this distinction is not relevant, *supervisor* and *controller* are often used interchangeably.

Supervisor synthesis

A correct-by-construction formal method that automatically synthesizes a *supervisor*. It involves the automatic generation, or synthesis, of a correct-by-construction controller model from a simple model of the to-be-controlled system and a model of the control requirements. This is also called controller synthesis or supervisory controller synthesis. Supervisor synthesis makes verification of the resulting supervisor model against the requirements from which it was synthesized superfluous. Validation of the resulting system being controlled by the controller (model) is still needed to ensure the specified requirements are indeed the desired requirements.

Uncontrollable event

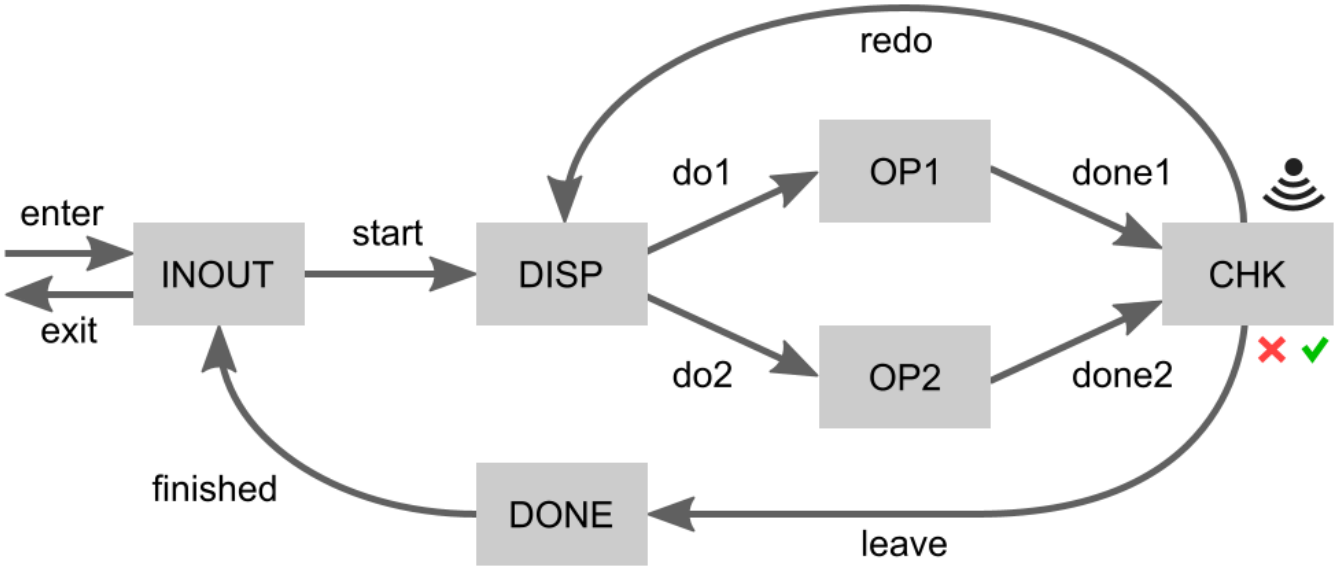
An event that operates autonomously, from the perspective of the controller. Such events are not controlled by the controller, which can thus not prevent them from occurring. For instance, events of a sensor could indicate that a button was pushed or released. And events of a movement limit sensor could indicate that a movement has reached the end position, or that the moving object is no longer at that position.

Uncontrolled system / plant

The uncontrolled system is the system 'as is', without any control. It is also called a plant in control theory. For instance, at a low abstraction level, this could be the individual actuators and sensors of a system. At a higher abstraction level, it could be a collection of controllers for subsystems.

1.3. Synthesis-based engineering example

To demonstrate the value of synthesis-based engineering, let's look at an example. The following figure illustrates an example manufacturing system that processes products:



Products *enter* at the *INOUT* place. From there production can *start* by moving them to the dispatcher (*DISP*). The dispatcher dispatches a product (*do1* or *do2*) to one of two operators (*OP1* or *OP2*) that perform the same operation. Once the operation is completed (*done1* or *done2*), the product is moved to the checker (*CHK*). The checker determines whether the operation has completed successfully or has failed. If it has failed, the system must *redo* the operation on that product. This may be repeated until the operation is successful. The product must then *leave* the processing loop, moving to *DONE*. It is then *finished* and moves back to *INOUT*. There it may *exit* the system.

The gray boxes indicate places where at most one product can be located at a time. The moving of products through the system is visualized by the labeled arrows in the figure. Each arrow corresponds to an actuator under the control of the controller. The controller can thus decide when to move products from one place to another. A sensor indicates the result of the check performed on processed products, indicating whether they are OK or not. This sensor works autonomously and is thus not controlled by the controller.

1.3.1. FIFO requirement

The example system, without any controller that controls it, already ensures that:

- Products that *enter* can only *start*, preventing them from exiting without having been processed.
- Products that failed processing must *redo* the operation.
- Successfully processed products must *leave* the operation area.
- Once a product passed *finished* it must *exit*, preventing it from being processed again.

For this example, we consider only a single requirement:

- Products must enter and exit the system in FIFO order.

That is, if one product enters earlier than another, it must also exit earlier.

Without additional control, the system does not satisfy this requirement, as it is possible for multiple products to enter the system and subsequently be processed concurrently. Then, if a later product finishes the operation earlier, or the earlier product requires rework, the later product may be done sooner and thus exit the system earlier. The controller must restrict the behavior of the system such that it satisfies the requirement. It can only do so by controlling the movement of products through the system.

The FIFO requirement is specified in natural language as a short and simple sentence. It can similarly be quite easily modeled, by tracking the order that products *enter* and *exit* the system. Each product that enters the system is given a unique identifying number, one higher than the previous product. As products *exit* the system, the identifier of the last product that exited the system is stored (*lastExitId*). When a product is about to exit the system, it is in the *INOUT* place. If the identifier of the current product on the *INOUT* place is given by *curId*, then the requirement can be formulated as:

- $curId = lastExitId + 1$

See the section on synthesis-based engineering in practice example section for how the example system and its requirement can be modeled in CIF.

1.3.2. Synthesis-based engineering

There are various ways to ensure the FIFO requirement holds. A silly solution is to never allow products to enter the system. As there are then no products in the system, products also never leave the system. Therefore, all (non-existent) products are in FIFO order. Another slightly more useful option is to only allow a single product to be processed at a time. This would however severely limit the productivity of the system. It is actually not that trivial to decide the exact conditions under which the products may move, while still ensuring the FIFO requirement is satisfied.

We can however automatically compute the conditions that must hold for each movement by applying supervisory controller synthesis. This computes for each movement the minimal restriction that must be applied to enforce the requirement. Through synthesis, we obtain a supervisory controller that restricts four movements:

1. Movement *done1* is only allowed if the following two conditions both hold:
    - Either there is no product at the *DISP* place, or it is a later product than at the *OP1* place.
    - Either there is no product at the *OP2* place, or it is a later product than at the *OP1* place.
2. Similarly, movement *done2* is only allowed if the following two conditions both hold:
    - Either there is no product at the *DISP* place, or it is a later product than at the *OP2* place.
    - Either there is no product at the *OP1* place, or it is a later product than at the *OP2* place.
3. Movement *start* is only allowed if one of the following two conditions holds:
    - At the *DISP* place, *OP1* place, *OP2* place, and *CHK* place, there is in total at most one product.
    - At the *DISP* place, *OP1* place, and *OP2* place, there is in total at most one product. There is also a product at the *CHK* place and the check indicates the product was successfully processed.
4. Movement *enter* is only allowed if less than four products are in the system.

But why are these the 'optimal' restrictions?

It is important to realize that:

A. If a product is checked and found to be successfully processed, it can only *leave*. It can not be reprocessed (*redo*). If a product is moved to *CHK* too early, a product that should exit the system before it can't overtake it anymore. This could lead to a violation of the FIFO property if another product that must exit earlier is for instance still being processed.

B. Only at most two products may be in the processing loop at any time. That is, at most at two of the *DISP*, *OP1*, *OP2* and *CHK* places there may be a product, at any time. This way, if a product keeps failing to be processed successfully, it can be redone over and over again, while the other product is at one of the operators. With three or more products in the processing loop, this is not possible. An exception to 'at most two products in the processing loop' rule is when a product has been checked and found to be successfully processed. Then, a third product may be present, as the successfully processed product can then leave the processing loop and at most

two products will remain in the processing loop.

Then the supervisor restrictions are quite logical:

- The first and second restrictions indicate when a product may move to be checked. These two restrictions follow directly from realization *A*. A product *X* may only be moved to be checked, if there is no product that must exit earlier. Obviously, moving a product to the checker is physically only possible if there is a product at an operator, as otherwise there is no product to move. Also, it is only physically possible to move a product to the checker there is not already a product at the checker, as each place can only hold one product. This leaves only the dispatcher and other operator as places to be checked. If there would be an earlier product at the dispatcher or other operator, such a product would not be able to overtake the product about to be moved to the checker, leading to a violation of the FIFO property. Hence, both restrictions have two conditions, one for the dispatcher and one for the other operator. Either there must be no product at those places, or it is later product.
- The third restriction indicates when a product may *start* processing. Either one of its two conditions must hold for the *start* movement to be allowed. This directly follows from realization *B*. The first condition follows from the 'at most two products in the processing loop' rule. At most one product may be in the processing loop for another to enter it. The second condition describes the exception to this rule. There may be two products in the processing loop if one of them is a successfully processed product about to leave the processing loop.
- The fourth restriction indicates when a product may *enter* the system. It only allows a product to enter if there are less than four products in the system. This means that the restriction ensures that at most four products are in the system at any time. Through realization *B* we know at most three products may be in the processing loop. Then only at most one of the *INOUT* and *DONE* places may contain a product, for a total of four products in the system. To understand why this is the case, consider the following:
  - A product could be at the *INOUT* place. But then no product must be at the *DONE* place. If there were a product at the *DONE* place, there would be products at the *INOUT*, *DONE* and *CHK* places. The product at the *CHK* place could then not move to the *DONE* place, as that already has a product. Similarly, the product at the *DONE* place could then also not move to the already occupied *INOUT* place. And the product at the *INOUT* place could then not move to the *DISP* place, as the processing loop is already maximally filled. This would mean no product could move anywhere. This kind of deadlock is prevented by the fourth condition.
  - A product could be at the *DONE* place. But then, by similar reasoning, no product must be at the *INOUT* place.

All of this is certainly a lot to consider! Would you have been able to figure all of this out by yourself? And how long would that have taken you? Considering this is only a simple example system with only one non-trivial requirement, it is clear that having some computer assistance when engineering a more realistic controller can be very useful.

1.3.3. Example benefits of synthesis-based engineering

Finally, let us consider some of the benefits of synthesis-based engineering as it relates to this example:

- Synthesis automatically computes the optimal control conditions. It should now be clear that this can **save a lot of effort**.
- Manually engineering the controller can be quite tricky. It could easily lead to mistakes if certain scenarios are not properly accounted for. For instance, a restriction could be missed, or one of them could be incorrect. Synthesis can thus also **reduce human error**.
- Through synthesis you only have to specify the requirement and synthesis automatically generates a correct-by-construction controller, from which you can automatically generate the implementation. For the simple to specify but difficult to implement example requirement, this allows you to **focus on *what* the controller should do** (the requirement), **rather than *how* the controller should do this** (the complex control conditions and their implementation).
- As an alternative to synthesis, we could apply formal verification on the system model to check whether the FIFO requirement holds. However, as the requirement does *not* hold on the system without a controller, we would get only a counter example representing a scenario indicating where the requirement does not hold. Likely, it would take several iterations and quite some thinking to manually arrive at the exact correct control conditions. **Compared to formal verification**, synthesis produces all the correct control conditions, **automatically and in a single iteration**.
- An engineer that develops the controller manually, may well impose severe restrictions to avoid much of the complexity of satisfying the FIFO requirement. The control conditions produced by synthesis however, are **minimally restrictive**. Products may enter the system, start processing, be processed in parallel, and leave the processing loop, whenever possible. This ensures the **maximum throughput** of the system can still be achieved.
- Synthesis-based engineering allows for a modular design. The various parts of the system, as well as the requirement, can be **modeled** separately. This makes it easy to adapt the system (model), to for instance allow products that do not require processing to bypass the processing loop. With minimal changes to the system model, and no changes to the requirement, a new supervisor can then be produced by the push of a button. This allows for **incremental development** of the system and its controller.

And again, consider that this is only a simple example system, with only a single requirement. Synthesis-based engineering has even more value when multiple, complex and related requirements need to be considered, or when controllers for many similar yet different systems need to be developed. See the section on benefits of synthesis-based engineering for further benefits of the approach.

Even though synthesis-based engineering has many benefits, companies should not underestimate how significantly different it is from traditional engineering. They should consider and manage the challenges particular to this engineering approach.

1.4. Synthesis-based engineering in practice

This section explains concretely how to use the CIF language and toolset to apply synthesis-based engineering of supervisory controllers. Before reading this section, please familiarize yourself with:

- The basics of the CIF language.
- The synthesis-based engineering approach.

Development process

The process to develop supervisory controllers using the synthesis-based engineering approach typically following several steps. We'll briefly discuss each of the steps and provide some practical guidance:

Modeling the events

Modeling the actions that can happen in the system as events.

Modeling the plant

Modeling the plant automata that represents the event behavior of the to-be-controlled system.

Modeling plant relations

Modeling the relations between the various plant automata.

Modeling the requirements

Modeling the requirements that restrict the behavior of the plant.

Marking

Dealing with marking of the plant and requirement automata.

Supervisor synthesis

Performing supervisor synthesis on the plant and requirements to automatically synthesize a supervisor.

Verification and validation

Verifying and validating that the synthesized supervisor controls the system correctly and as desired.

Controller implementation

Implementing the validated supervisory controller using automatic code generation.

Advanced topics

Furthermore, the following more advanced information is available:

Incremental controller development

Explains how to incrementally develop your controller to prevent commonly encountered issues when applying synthesis-based engineering.

Resolving issues with too limited behavior

Explains how to resolve issues with too limited controlled system behavior, for instance due to conflicting requirements, revealed through synthesis or validation.

Supervisor synthesis performance

Explains how to resolve performance and memory issues for supervisor synthesis.

Non-monolithic supervisor synthesis

Explains how to incrementally develop your controller to prevent commonly encountered issues

when applying synthesis-based engineering.

Practical example

For how this process can be used in practice, based on an example, see:

- Synthesis-based engineering example
- Synthesis-based engineering in practice example

1.4.1. Development process

Modeling the events

The first step to apply synthesis-based engineering in practice is to identify the actions that can happen in the system, and model them as controllable and uncontrollable events.

Controllable events are under the control of the controller. They are outputs of the controller and inputs for the system being controlled. The controller determines when these events occur. Typical examples include turning an actuator on or off, or enabling or disabling a sub-system.

Uncontrollable events are outside of the control of the controller. They are inputs for the controller and outputs of the system being controlled. The controller can not prevent such events from occurring in the system. Typical examples include a sensor going on or off, such as for a button push sensor or a movement limit sensor.

The events can be modeled in CIF as follows:

```
uncontrollable u_button_pushed, u_button_released;
controllable c_turn_motor_on, c_turn_motor_off;
```

Controllable events are by convention given names starting with `c_`, while uncontrollable events start with `u_`.

For channels, a special kind of events, see the CIF language tutorial:

- Channels
- Dataless channels
- Combining channel communication with event synchronization

The next step in the process is to model the plant.

Modeling the plant

After modeling the events, the plant needs to be modeled. It represents the uncontrolled system, the system 'as is' without the controller.

Typically, for low-level controllers, start with a plant automaton per sensor and actuator. For the common case of digital sensors and actuators, model the automata with two locations, one where the sensor or actuator is off, and one where it is on. Which location should be the initial location depends on the specific sensor or actuator. Digital sensors can go on and off, and as such have two associated uncontrollable events. Similarly, digital actuators can be turned on or off, and have two associated controllable events.

Here are some examples of typical plant automata for low-level sensors and actuators:

```
plant Button:
  uncontrollable u_pushed, u_released;

  location Released:
    initial; marked;
    edge u_pushed goto Pushed;

  location Pushed:
    edge u_released goto Released;
end

plant Lamp:
  controllable c_on, c_off;

  location Off:
    initial; marked;
    edge c_on goto On;

  location On:
    edge c_off goto Off;
end
```

The events that belong to a specific sensor or actuator are typically placed within the corresponding automaton. Other events are often placed outside the automata.

See a later step for how to deal with marking. The CIF language tutorial has lessons on using variables, guards and updates.

The next step in the process to apply synthesis-based engineering in practice is to model plant relations.

Modeling plant relations

After modeling the plant, for instance the individual sensors and actuators, the relations between the plant automata are to be considered. These relations should be physical relations, representing behavioral restrictions present in the actual uncontrolled system.

For instance, consider a movement with two limit sensors:

```
plant UpSensor:
  uncontrollable u_on, u_off;

  location Off:
    initial; marked;
    edge u_on goto On;

  location On:
    edge u_off goto Off;
end

plant DownSensor:
  uncontrollable u_on, u_off;

  location Off:
    initial; marked;
    edge u_on goto On;

  location On:
    edge u_off goto Off;
end
```

Most likely the two sensors physically can't both be on at the same time. The individual plant automata of the two digital sensors however, can both be in their `On` states, as they are not yet in any way related. The easiest way to specify such a relation is to use a state plant invariant:

```
plant invariant not (UpSensor.On and DownSensor.On);
```

Alternatively, you may combine multiple plants into a single plant. To merge some plants, manually compute/model the product of the plants, and remove the original plant automata. Then, to express the relationship, remove the behavior that is not physically possible. However, typically using a plant invariant is easier.

An alternative physical relationship, is the relation between sensors and actuators. In such cases, the relationship with the sensor(s) can usually be added directly to the actuator plant(s).

By correctly incorporating all the physical restrictions present in the actual system, the tools can use this knowledge during synthesis. Essentially, by modeling the physical relations/restrictions, the uncontrollable events are enabled in much less (combinations of) locations of the plant automata. This means that the requirements are much less likely to block uncontrollable events.

In other words, the modeled relationships of the plants restrict the behavior of the plant automata. However, these restrictions are also present in the physical system. Hence, without modeling such relationships, the plant model has more behavior than the physical system. Once the plant relations are correctly modeled, you may assume this relationship in the requirements, meaning you may assume that certain uncontrollable events can physically not occur in certain locations. The tools

will then have enough knowledge of the system to come to the same conclusions.

For instance, assume a certain sensor signal can only occur when the corresponding actuator is enabled. A movement limit sensor may for instance only be able to go on once a movement completes, which in turn can only happen by enabling the corresponding movement actuator. Modeling this relation ensures that 'blocking' such sensor signals in the requirements, when the actuator is off, is no longer considered 'illegal' behavior.

The next step in the process to apply synthesis-based engineering in practice is to model the requirements.

Modeling the requirements

After modeling the plant and plant relations, the requirements should be modeled as well.

The hardest thing about modeling the requirements, is that you have to think in restrictions, rather than in use cases. So, rather than 'first do this, then do that, then do that or that other thing, etc', you should think 'this or that is only allowed if/after this or that other thing'. Requirements should be as small and orthogonal as possible.

Event-based requirements are modeled as requirement automata. The simplest event-based requirements have only two locations, and form a loop of only two edges. Here is a typical example requirement that controls the plants from the section on modeling the plant. It ensures that the lamp is on while the button is pushed, and off while it is released:

```
requirement LampOnWhileButtonPushed:
  location Released:
    initial; marked;
    edge Button.u_pushed goto Pushed;
    edge Lamp.c_off;

  location Pushed:
    edge Button.u_released goto Released;
    edge Lamp.c_on;
end
```

We can also model the requirements in a more state-based manner (referring to locations of automata) or data-based manner (referring to locations of automata, as well as using variables, guards, updates, and invariants), which is often shorter and simpler. The requirement above can be modeled in a state-based manner using state/event exclusion requirements as follows:

```
// Lamp on only while button is pushed.
requirement Lamp.c_off needs Button.Released;
requirement Lamp.c_on  needs Button.Pushed;
```

Having requirements block uncontrollable events can easily lead to unnecessarily restricting too

much of the system behavior. As mentioned in the section on modeling plant relations, correctly modeling such relations makes this easier.

Generally, it is better to as much as possible use requirements that are pure restrictions. That is, use state-based requirements (mutual state exclusion and state/event exclusion requirements) instead of event-based requirements (requirement automata), where applicable. Requirement automata may introduce additional state, which can lead to reduced performance. Using pure restriction requirements you are also less likely to unnecessarily restrict too much of the system behavior.

The CIF language tutorial has lessons on using variables, guards and updates.

Keeping required behavior possible

Requirements that rule out unwanted behavior do not by themselves ensure that useful behavior remains possible. Even with marking, synthesis only needs to preserve the possibility of reaching some marked state. Add a reachability requirement when a particular capability must remain available and is not already ensured by marking. For example, a bridge may need to remain able to open for ships to pass, and able to close for road traffic to pass. Separate reachability requirements for the open and closed states ensure that neither capability can be lost merely because the other one suffices to reach a marked state. Other useful goals include returning a machine to its idle state after an operation or recovering to an operational state after a fault. These requirements keep paths to the goals available; they do not force the system to take those paths.

Use `from` when a capability is needed only in certain source states, and `avoid` or `stay` when the path to the goal must respect an additional restriction. For example, it may be necessary to keep an operational state reachable without passing through maintenance. A conditional requirement can be satisfied by removing all its source states from the controlled system. If those states must remain reachable as well, add a basic reachability requirement with the source predicate as its goal. For a bridge, `@requirement:reachable(closed, from: open)` keeps closing possible from the remaining open states, while an additional `@requirement:reachable(open)` ensures that opening itself remains possible. This requires some state satisfying the source predicate to remain reachable from every controlled state, rather than retaining every individual source state of the uncontrolled system.

Use `bound` when the number of model transitions needed to reach the goal matters. Choose the bound with the modeled operation and its possible starting states in mind: different states may need different numbers of transitions, and every state to which the requirement applies needs a permitted path within the bound. For example, some starting states may require preparatory actions before the operation itself can begin. Count those actions as well if they are separate transitions in the model. A bound limits the length of a path that must remain available; it does not impose a time deadline or require every execution to reach the goal within that many transitions. Changing the level of detail of the plant model may change the appropriate bound.

If no meaningful limit on the number of transitions is known, first model the requirement without a bound. When adding a bound, check that the resulting supervisor still permits the intended operations and source states: a smaller bound can remove behavior or lead to an empty supervisor. The data-based synthesis tool's debug output can help explain the effect of a bound, but it is important to always interpret a reported minimum bound together with its stopping reason and the current controlled behavior. A fixed point reached before all source states are covered does not demonstrate that those source states can reach the goal within the reported minimum bound.

A bound can also be used for regression checking: a model update that unintentionally adds required steps may make the goal unreachable within the expected bound. Successful synthesis alone is not sufficient for this check, since synthesis may satisfy the bound by removing behavior or source states. As part of the regression check, you should therefore also verify that both the intended behavior and the intended source states remain reachable.

The next step in the process to apply synthesis-based engineering in practice is to deal with marking.

Marking

After modeling the plants and requirements, marking should be considered. Every automaton, whether plant or requirement, must have at least one marked location. A marked location is a location that indicates a safe, stable, or resting state. Synthesis will guarantee that a marked location can always be reached, thus ensuring a form of liveness.

Physical systems typically keep operating, repeating their behavior. Therefore, in practice, the entire system can often be brought back to the initial state. As such, it is then typically enough to make the initial locations marked.

An exception is automata that have some kind of initialization behavior/sequence. For such automata, make the first location that is part of the loop after the initialization sequence, a marked location. The locations from the initialization sequence can no longer be reached after initialization is finished. The first location after that initialization sequence is part of the 'normal' behavior and can be seen as the initial location of the behavior after initialization.

Note that marking every location reduces the value of supervisor synthesis, as it essentially disables its non-blockingness guarantee. This may hide issues related to deadlocks and livelocks.

For advanced uses of marking, see the CIF language tutorial section on marker predicates. Beyond marking, CIF also supports reachability requirments, which may be combined with marking to get stronger liveness guarantees.

The next step in the process to apply synthesis-based engineering in practice is to synthesize a supervisor.

Supervisor synthesis

Having modeled the plants and requirements, the supervisory controller can automatically be synthesized. Supervisory controller synthesis (or simply supervisor synthesis) automatically generates a *supervisor*, from the models of the uncontrolled system (plant model) and control requirements (requirements model).

The synthesized supervisor is correct-by-construction, satisfying various properties. It is *safe* (satisfies all the requirements), *controllable* (limits only controllable events, not uncontrollable ones), is non-blocking (does not block, a form of liveness), and is maximally permissive (imposes no

more restrictions than is necessary).

CIF supports supervisor synthesis through the following tools:

- Data-based supervisory controller synthesis tool
- Event-based supervisory controller synthesis tool

The data-based synthesis tool is generally more efficient and supports a larger subset of CIF language concepts. It is therefore recommended over the event-based synthesis tool.

The documentation of these tools provide further details on how to use them.

The next step in the process to apply synthesis-based engineering in practice is to perform verification and validation.

Verification and validation

After applying supervisor synthesis it is time to analyze the resulting supervisor model, using verification and validation, and adapt the plants and requirements if any issues are found.

Verification

Verification to ensure that the synthesized supervisor satisfies the requirements that were used for synthesis is superfluous, as the synthesized model is correct-by-construction. However, verification may still be needed for additional requirements that are not yet supported by synthesis, such as stronger liveness requirements and timed requirements.

The synthesized supervisor model should be checked for various properties using the controller properties checker tool. Further verification may be performed by transforming the supervisor model to formats of external verification tools, such as mCRL2 and UPPAAL, and then using such tools to perform the actual verification.

Validation

Furthermore, the supervisor should be validated to ensure it behaves as intended. The specified requirements could not be the desired requirements, as they could for instance be wrongly specified or too strict, resulting in the system being controlled by the controller exhibiting unwanted or insufficient behavior.

The CIF simulator can be used to simulate CIF specifications. Especially when combining this with SVG visualization and interactive simulation, it is a very powerful way to validate whether the supervisory controller controls the system as intended. This may for instance reveal that additional requirements are needed, or existing requirements need to be adapted.

Next steps

In case issues are found through verification or validation, these need to be addressed. Typically this involves changes to either the plant model or requirements model. After such changes, the supervisor can be re-synthesized automatically. Changes can be made iteratively, until confidence

in the correctness of the controller is high enough.

The next step in the process to apply synthesis-based engineering in practice is then to implement the supervisory controller.

Controller implementation

Once a supervisory controller has been synthesized, verified and validated, it is time to implement it. The control software is typically implemented using a programming language, such as PLC code for a PLC platform, or Java or C++ code for an industrial PC. This may for instance be done in-house within the company, by different teams or departments, or by an external supplier.

While manual implementation is possible, the code is often automatically generated from the controller model. This automatically produces correct-by-construction code by generating the code from the correct-by-construction controller model. Automation prevents the kinds of subtle mistakes that humans make when they manually implement something, ensuring consistency between the controller model and implementation. Automation also improves efficiency. If the controller model is changed, with the push of a button a new correct-by-construction implementation can again quickly be generated.

CIF supports code generation through the following tools:

- CIF PLC code generator: for PLC programming languages
- CIF code generator: for various other programming languages

The documentation of these tools provide further details on how to use them.

Once code has been generated, it should be tested on the actual system. This may reveal additional issues. For instance, the behavior of the system may not be properly captured in the plant model, or the inputs and outputs of the system may be incorrectly connected to the controller.

With a working implementation of the controller, this concludes the process to apply synthesis-based engineering in practice.

1.4.2. Advanced topics

Incremental controller development

When developing a controller, it is often best to start with just a small part of the system.

You first develop a controller for that small part, and verify and validate it to make sure it works properly. Then, you extend the controller to work for a larger part of the system, and you keep repeating this until the controller controls the entire system. That is, you go through the entire development process (including among others specification, synthesis, verification/validation, implementation) each time. And then you increase the part of the system covered by the plants and requirements for each iteration. Typically, for low-level controllers you should start with no more

than a few sensors and actuators, with their corresponding events.

This incremental development approach makes it easier to for instance figure out why your controller doesn't behave as expected, which of your requirements are conflicting, or for what other reason a supervisor can't be synthesized.

Resolving issues with too limited behavior

Supervisor synthesis always produces correct-by-construction supervisors, based on the plant and requirements models that you provide as input. However, if you for instance provide too restrictive or conflicting requirements, or forget to model the plant relations, you may not get the desired controlled system behavior. Such issues are often found during verification and validation.

Another way this may manifest itself, is by supervisor synthesis producing an 'empty supervisor' error. This means that synthesis has determined that no supervisor can ever safely satisfy the requirements that you specified.

However, the resulting supervisor supervisor doesn't have to be 'empty'. Synthesis could also have restricted so much of the behavior that little behavior remains, in order to satisfy the (conflicting) requirements you provided. The resulting supervisor could for instance have only a few states, but it may also have thousands or millions of states, while still missing important system behavior.

In such cases, where the resulting supervisor is not what you expect or desire, you need to go back to your plants and requirements. Here are some hints to resolve this kind of problems:

- Try to use an incremental development approach. This ensures that if you encounter problems with missing behavior, you can be reasonably sure the problem is in the part you added since the last working version. If on the other hand you put the entire system in your model at once, and you get for instance an 'empty supervisor' error, it is much more difficult to track down the cause.
- Try to as much as possible use requirements that are pure restrictions.
- Make sure you have at least one marked location per automaton (plant as well as requirement automata). Usually, marking the initial location is sufficient. See also the section on dealing with marking.
- Make sure your initial and marked locations are consistent between all automata. For instance, if in a plant you can initially only push a button, and then release it, but if in a requirement automaton you must first release it before it can be pushed, you are likely to get an empty supervisor.
- Make sure your requirements don't restrict the system too much. Be especially careful with blocking uncontrollable events in requirement automata.

  For every uncontrollable event in the alphabet of a requirement automaton, make sure that the requirement does not block the uncontrollable event. You can look at the plants, to see when the uncontrollable event is possible. Then you may ask yourself, for each location of the

requirement, in which locations of the plant you can be at the same time. For such plant locations, you should check whether the uncontrollable event is possible. It should then also be possible in the requirement. There are two ways to solve the blocking of an uncontrollable event by a requirement:

  - The first solution is to correctly model the relations between the plants. For further details, see the section on modeling plant relations.
  - The second solution is to add self loops in the requirement to allow the uncontrollable event that was previously disabled by the requirement.

  The first solution is recommended, but it does not always apply. If it is not possible to use the first solution, or if you already applied the first solution and still have the problem, you could use the second solution. However, never blindly add self loops. Always check that this gives you the desired behavior!

- Try to synthesize a supervisor with a subset of the requirements, to find out which requirement (or combination of requirements!) is causing the lack of behavior.

Supervisor synthesis performance

There are several ways to altogether avoid performance and out-of-memory issues when applying supervisor synthesis:

- Use the data-based synthesis tool rather than the event-based synthesis tool, as the former has much better performance.
- Rather than synthesizing a single monolithic supervisor, employ non-monolithic synthesis to make use of the inherent structure of your system.

However, if you still suffer such issues, consider the following to resolve them:

- See the page of the Eclipse ESCET general toolkit documentation on resolving performance and memory problems. In particular, make sure to give Java more memory. This should be the first thing you check.
- If possible, use requirements that are pure restrictions, as they only make the supervisor smaller. If you introduce memory (e.g. counters), or other forms of sequencing (e.g. to specify the order of controllable events), that usually increases the size of the supervisor. If possible, use requirements with only one location, or even better, use requirement invariants. For more information, see the State (exclusion) invariants, State/event exclusion invariants and Invariant kinds sections of the CIF language tutorial.
- Try to separate your requirements as much as possible, to keep them simple. Also, leave out any events from a requirement automaton that are not relevant to that specific requirement.
- You can try to restrict more of the system, by adding more requirements, to get a smaller supervisor (with less behavior).
- Avoid adding plants that you don't at all use in the requirements. For instance, don't add a

button plant automaton if you don't use the button.

- Try to avoid duplicate requirements that enforce the same behavior in different ways.
- Check out the performance section of the data-based synthesis tool documentation.

Non-monolithic supervisor synthesis

A known concern for supervisory controller synthesis is its scalability. While a supervisor may be synthesized for smaller systems in seconds, for larger and more complex systems this may take considerably more time. Employing the proper techniques for the given situation is essential to mitigate this concern.

Often, rather than synthesizing a single monolithic supervisor, multiple supervisors should be synthesized for different parts of the system, making use of the inherent structure of the system itself. See for instance the CIF multi-level splitter. Combining this with abstraction, higher-level supervisors can still be synthesized on top of lower-level supervisors, scaling to very large systems.

Still, multiple syntheses will need to be performed. If you encounter scalability issues for some of them, divide the system further, or resolve the performance problems in another way.

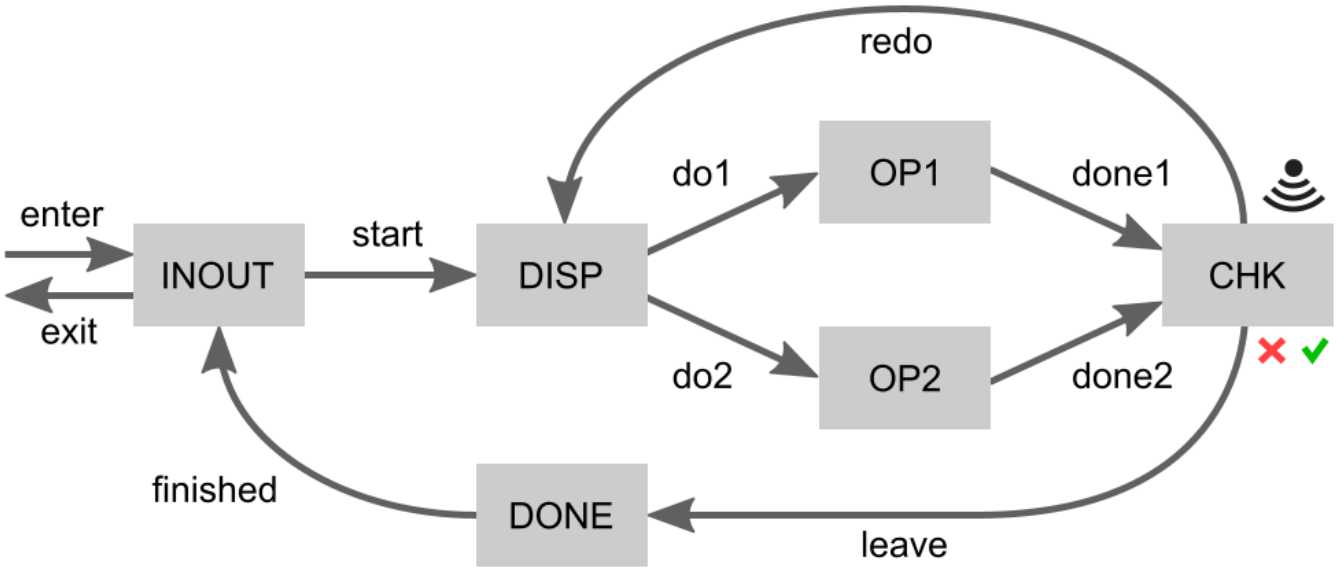
1.4.3. Synthesis-based engineering in practice example

CIF supports the entire development process for synthesis-based engineering of supervisory controllers. The steps involved are described in the section on synthesis-based engineering in practice. However, here we focus on specification, supervisory controller synthesis, simulation and code generation.

We consider how a controller can be developed for the synthesis-based engineering example. This example is one of the many CIF examples. See the CIF examples section for how to import them into your Eclipse ESCET IDE, to experiment with them yourself. After importing the examples project into your IDE, you can find this example in the project's `synthesis/fifo` folder.

Plant specification

First, we'll specify the plants (file `fifo.plants.cif`). Consider again the following figure that visualizes the example system:



Events

Each of the arrows indicates a movement of products. We can model each arrow as a controllable event that the supervisor can control. The checker (*CHK*) may indicate that a product was processed successfully or that it failed, which we can model as uncontrollable events. This leads to the following event declarations:

```
controllable c_enter;
controllable ProductId c_start;
controllable ProductId c_do1;
controllable ProductId c_do2;
controllable ProductId c_done1;
controllable ProductId c_done2;
uncontrollable u_success;
uncontrollable u_failure;
controllable ProductId c_redo;
controllable ProductId c_leave;
controllable ProductId c_finished;
controllable c_exit;
```

Product identifiers

To be able to express the example's FIFO requirement, we must keep track of product identifiers (ids). A product may *enter* the system, after which it gets its id. As products move through the system, their id is passed along. Hence, most of the events are channels that communicate a `ProductId`. This is a custom type, defined as follows:

```
const int MAX_NR_OF_PRODS = 5;
type ProductId = int[0 .. MAX_NR_OF_PRODS - 1];
```

CIF can only perform synthesis on specifications where types have a finite domain. We therefore define a maximum number of products (`MAX_NR_OF_PRODS`) that may be in the system. Product ids are then integer numbers in the range `[0 .. MAX_NR_OF_PRODS - 1]`, with both bounds being inclusive.

This allows each product in the system to have a unique id.

The *INOUT* place

Each of the system's places that can holds a product is modeled as a plant automaton.

First we model the *INOUT* place:

```
plant INOUT:
  disc ProductId nextId = 0;
  disc ProductId curId = 0;
  disc ProductId lastExitId = MAX_NR_OF_PRODS - 1;
  disc int[0..MAX_NR_OF_PRODS] cnt = 0;

  location Idle:
    initial;
    marked;
    edge c_enter when cnt < MAX_NR_OF_PRODS do curId := nextId, nextId := (nextId + 1)
mod MAX_NR_OF_PRODS, cnt := cnt + 1 goto NewProduct;
    edge c_finished? do curId := ? goto FinishedProduct;

  location NewProduct:
    edge c_start!curId do curId := 0 goto Idle;

  location FinishedProduct:
    edge c_exit do lastExitId := curId, curId := 0, cnt := cnt - 1 goto Idle;
end
```

The automaton has several variables:

- Variable `nextId` keeps track of the product id to use for the next product that enters the system. The first product to enter the system gets id `0`.
- Variable `curId` represents the product id of the product that is currently located at the *INOUT* place. However, its value is irrelevant when there is no product at the *INOUT* place.
- Variable `lastExitId` keeps track of the last product that exited the system. Given that products must enter and exit in FIFO order, and that the first product to enter gets id `0`, `lastExitId` is initialized to the largest possible product id.
- Variable `cnt` counts the number of products currently in the system. As initially there are no products in the system, it is initialized to `0`. The count is used to ensure `MAX_NR_OF_PRODS` can be honored.

The automaton also has several locations. Initially the *INOUT* place is `Idle`, as it has no product.

A product may only enter (by event `c_enter`) if the maximum number of products is not yet exceeded (`cnt < MAX_NR_OF_PRODS`). The product then gets assigned the next product id (`curId := nextId`), it being a newly entered product currently located at the *INOUT* place. As the next product id has then been used, it will be incremented by one to ensure the next product again gets a unique product id (`nextId := (nextId + 1)`). Given that the `ProductId` type only allows a finite number of

ids, we loop around to avoid overflow (`mod MAX_NR_OF_PRODS`). We also update the number of products in the system (`cnt := cnt + 1`). The automaton then proceeds to its `NewProduct` location (`goto NewProduct`).

In the `NewProduct` location, processing of a product may *start* (event `c_start`) by sending it (`c_start!`) to the dispatcher (*DISP*). The product id is sent along (`!curId`) with this movement. After moving the product to the dispatcher, the *INOUT* place no longer holds a product (`goto Idle`). The product id is reset to zero (`curId := 0`). This is optional, but keeps the state space smaller, leading to more efficient synthesis.

While `Idle` the *INOUT* place may receive a *finished* product (`c_finished?`) from the *DONE* place. The product id of the currently present product is then updated to that of the received product (`curId := ?`). The automaton then proceeds to the `FinishedProduct` location.

There the product may exit (event `c_exit`) the system. It then becomes the last product to have exited the system (`lastExitId := curId`). There is then no longer a product at the *INOUT* place. Again the product id is reset to zero (`curId := 0`) to keep the state space smaller for efficient synthesis. As a product has left the system, the counter is also updated (`cnt := cnt - 1`).

The `Idle` location is also a marked location. This specifies that the *INOUT* place must always be able to become idle again. Through similar markings in the other places also the system as a whole must always be able to become idle again.

The other places

For brevity, we'll not explain the remaining places in as much detail as the *INOUT* place. We'll discuss each automaton for each place briefly:

- The dispatcher (*DISP*) receives products (`c_start?`) from the *INOUT* place. It forwards them to either the first (*OP1*) or second (*OP2*) operator, with `c_do1!curId` or `c_do2!curId`, respectively.
- The operators (*OP1* and *OP2*) simply receive a product from the dispatcher (*DISP*) and forward it to the checker (*CHK*), after some processing.
- The checker (*CHK*) receives a product from one of the operators (`c_done1?, c_done2?`). It then determines whether the product was successfully processed (`u_success`) or processing has failed (`u_failed`). It forwards successfully processed products to the *DONE* place by `c_leave`, while failed products are sent back to the dispatcher (*DISP*) by `c_redo`.
- The *DONE* place simplify forwards products from the checker (*CHK*) to the *INOUT* place.

The FIFO requirement

With the plants specified, we specify the requirement (file `fifo.plants_and_requirements.cif`). We specify it in a separate file, to allow using the plant model for both synthesis and simulation, as described later in this section.

First, we import into this file the entire plant specification:

```
import "fifo.plants.cif";
```

The requirement was given in natural language as:

- Products must enter and exit the system in FIFO order.

We can easily model it as follows, using a requirement invariant:

```
requirement FIFO: INOUT.FinishedProduct => INOUT.curId = ((INOUT.lastExitId + 1) mod
MAX_NR_OF_PRODS);
```

If the *INOUT* place has a finished product (is in its `FinishedProduct` location), then a product is about to exit the system. We know the product id of the last product that exited (`INOUT.lastExitId`) and the product id of the product currently situated at the *INOUT* place (`INOUT.curId`). Products that enter the system get an id that is one higher than the previous product that entered (modulo the maximum number of products). The FIFO property can thus be ensured by requiring that when a product exits, it also has an id one higher than the last product that exited (again modulo the maximum number of products). From this, requirement `FIFO` follows directly.

Performing synthesis

We can automatically compute a supervisory controller by applying supervisory controller synthesis. For this, we'll use the CIF data-based synthesis tool.

To be able to perform synthesis with the push of a button, a script is provided (file `do1_synthesize.tooldef`):

```
from "lib:cif" import *;

mkdir("generated", force=true);

cifdatasynth(
    "fifo.plants_and_requirements.cif",
    "--forward-reach=true",
    "-m debug",
    "-o generated/fifo.synthesized.cif",
);
```

It first imports the CIF tools. Then it ensures that directory named `generated` exists. It is thus created if it does not yet exist. Lastly, it invokes the CIF data-based synthesis tool on the file that contains the plants and requirements. It configures some options. Forward reachability is enabled for simpler resulting control conditions. It also enables debug output to be printed to the console, allowing to see what synthesis has done. Finally, it specifies that the synthesis result is to be saved to the `fifo.synthesized.cif` file in the `generated` directory.

To execute the script, right click it an choose **Execute ToolDef** or select the file and press `F10`.

Simulation model

To validate the system controlled by the synthesized supervisor, it can be useful to simulate it. We therefore specify a simulation model (file `fifo.simulation.cif`).

This model first imports the synthesized supervisor:

```
import "generated/fifo.synthesized.cif";
```

It then specifies an SVG image to be used for visualization of the system:

```
svgfile "fifo.svg";
```

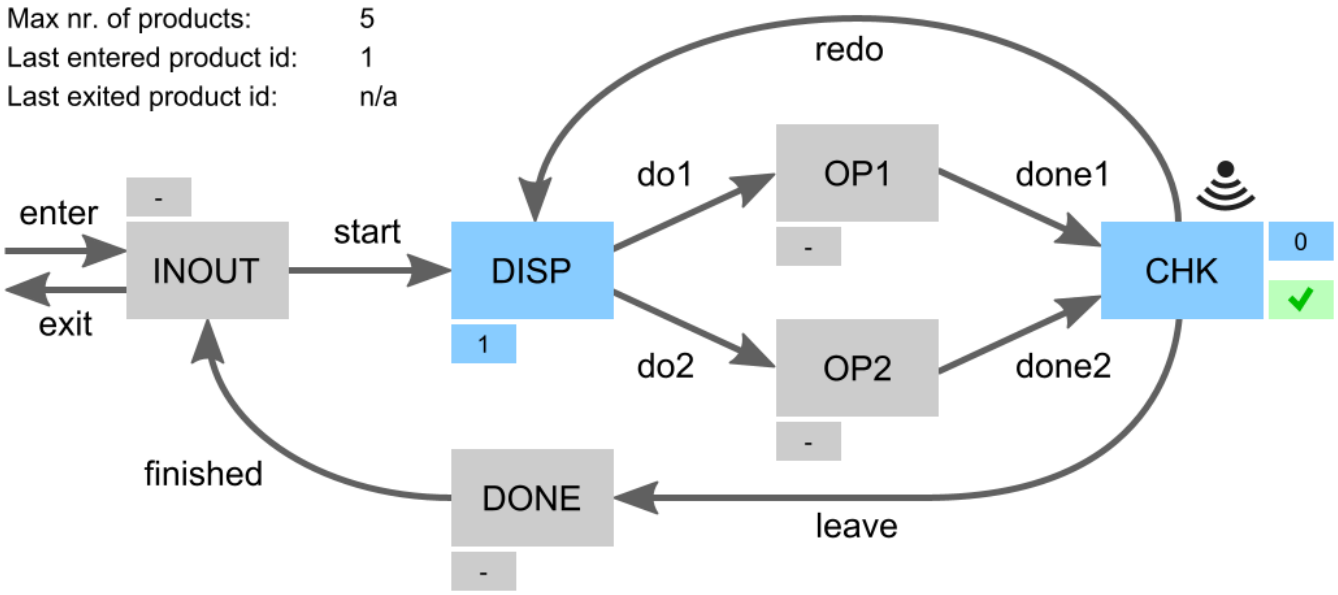
The bulk of the specification consists of CIF/SVG output mappings that map the state of the plant model to properties of elements of the SVG image. Some of them ensure that text labels have the correct text, others ensure that boxes have the correct fill color, etc. Here are some examples:

```
svgout id "max-nr-of-prods" text             value <string>MAX_NR_OF_PRODS;
svgout id "inout-cur-txt"   text             value if INOUT.Idle: "-"        else
<string>INOUT.curId end;
svgout id "inout"           attr "fill"      value if INOUT.Idle: COLOR_IDLE else
COLOR_BUSY          end;
svgout id "chk-rslt"        attr "visibility" value if CHK.Idle:   "hidden"   else
"visible"           end;
svgout id "chk-rslt"        attr "fill"      value switch CHK:
                                                    case Idle:    COLOR_IDLE
                                                    case Busy:    COLOR_BUSY
                                                    case Success: COLOR_SUCCESS
                                                    case Failure: COLOR_FAILURE
                                                  end;
```

By using CIF/SVG input mappings, certain elements of the SVG image can be clicked to trigger events in the model. Here are some examples:

```
svgin id "enter"    event c_enter;
svgin id "start"    event c_start;
svgin id "finished" event c_finished;
svgin id "exit"     event c_exit;
```

The following figure shows a potential visualization of the state of the system during a simulation:



A place is gray if there is no product. The smaller box next to it is then also gray and indicates `-`, for no product. An occupied place is blue, with its smaller box indicating the product id of the product that is present.

For the checker (*CHK*) a second small box is present. It is hidden if there is no product at the checker. If a product is present for which a check has not yet been done, then the box is blue and has a question mark. If the check has completed and the product was processed successfully, then the box is green and has a tick mark. If processing the product failed, then the box is red with a cross mark.

At the top left some additional information is provided. It indicates the maximum number of products that may be in the system, as configured via `MAX_NR_OF_PRODS`. It further indicates the product id of the last product that entered the system, or `n/a` if no product has entered the system yet. Finally, it indicates the product id of the last product that existed, as indicated by `INOUT.lastExitId`, or `n/a` if no product has exited thus far.

It can not be determined from the plant whether any products have entered or exited thus far, nor what is the product id of the last product that entered. To ensure this information is available to be used in the CIF/SVG output mappings, a monitor automaton is added to the simulation model that keep tracks of this information:

```
automaton monitors:
  disc bool anyInput = false;
  disc bool anyOutput = false;
  disc ProductId lastEnterId = 0;

  location:
    initial;
    edge c_enter do  anyInput := true, lastEnterId := INOUT.nextId;
    edge c_exit  do anyOutput := true;
end
```

As products enter (event `c_enter`) or exit (event `c_exit`) the variables are updated as needed. The

variables are used in the CIF/SVG output mappings.

The labeled arrows in the figure can be clicked to trigger their corresponding events. For example, clicking the arrow labeled with `enter` triggers event `c_enter`. When an arrow is clicked and the corresponding event is not enabled, a warning will be printed to the console.

Simulating the supervised system

Similar to having a script to perform synthesis, a script is present to perform simulation (file `do2_simulate.tooldef`):

```
from "lib:cif" import *;

cifsim(
    "fifo.simulation.cif",
    "-i svg",
    "--frame-rate=30",
);
```

This script also imports the CIF tools. It then starts the CIF simulator. It configures some of the simulator's options, among others to indicate the simulation model to simulate.

Start the script as before. Simulation will start and show the visualization.

Manually modeled supervisor

The example project also contains two other scripts. They can be used to check whether a manually modeled supervisor (file `fifo.manually_modeled_supervisor.cif`) has the same behavior as the synthesized supervisor. The first script (file `do3_chk_cif.tooldef`) performs this check using various CIF tools only. The second script (file `do3_chk_mcrl2.tooldef`) performs the same check using mCRL2.

Normally, one would not manually model the supervisor, and thus also not perform such checks. However, for this example we include them, as they may prove illustrative.

Code generation

Finally, from the synthesized supervisor an implementation of the controller may be automatically generated. CIF has tools to generate code for several programming languages.

1.5. Challenges in applying synthesis-based engineering

A synthesis-based engineering approach has many advantages over a more traditional engineering approach. However, there can be challenges when embedding such a new approach into industrial practice. It is essential to be aware of them, and manage them explicitly.

Most of the challenges that apply to a synthesis-based engineering approach also apply to other model-based engineering approaches, including verification-based engineering.

The following challenges are discussed:

- Change in way-of-working
- Tool support

1.5.1. Change in way-of-working

The use of model-based engineering, the modeling of behavioral specifications and control requirements, and the use of formal techniques such as supervisory controller synthesis, requires a certain mindset, knowledge and skills. It is important that personnel with the appropriate knowledge and expertise is present in a company. Having a team of properly trained and experienced experts that can assist with and steer the introduction of new techniques is essential. If a company does not have such experts, they could hire them. However, training and retraining for (part of) the existing personnel is often also required.

Furthermore, model-based engineering partly also requires a different way of working compared to traditional engineering approaches. It is important to understand the effects on the company's development process, as well as its culture. The various pros and cons must be evaluated, and any impediments must be identified and addressed.

A transition like this will not happen in a day. Sharing experiences with other parties that have gone through a similar transition and/or are going through one can be of great benefit. Another way to reduce the risks is contracting an external party to help guide the process.

Furthermore, risks can be reduced by step by step introducing the changes to the development process, introducing more and more elements of model-based, verification-based and synthesis-based engineering. This way models become more and more leading, throughout the development process. For instance, you could follow these steps:

1. **Start modeling (the requirements)**: Increase the quality of requirements by specifying them formally in a model-based way, during early development phases. This can already be combined with for instance simulation, to produce unambiguous specifications, leading to less mistakes and reduced rework. In this first step, the resulting requirements can still be put in a document and implemented manually.
2. **Models as single source of truth**: Formally but manually specify the controller model in a model-based way, based on the formal requirements. From the model, automatically generate the controller code. This is a step towards making the model the single source of truth.
3. **Embrace formal methods**: In this step, employ more formal methods to go beyond simulation and testing. Use formal methods that have more guarantees on completeness. For instance, use formal verification to guarantee that all specified requirements are satisfied in every conceivable situation. At the end of this step, you could fully adopt model-based and verification-based engineering.
4. **Adopt synthesis-based engineering**: Use supervisory controller synthesis, and fully adopt synthesis-based engineering.

1.5.2. Tool support

Synthesis-based engineering requires tool support to model plants and requirements, to synthesize supervisors, perform simulation, generate code, etc. Given that automation and computer-aided design are core principles, this is simply not feasible without appropriate tool support. Companies should consider various aspects regarding the tools they use, such as the following:

- Tools ideally support as much of the development process of supervisory controllers as possible.
- Consider how to integrate the synthesized supervisors into the system.
- Consider how active the community around the tool is.
- Consider whether commercial support is available.

The selected tools should also be used in the right way. For instance, naively applying synthesis and trying to obtain a single monolithic supervisor for larger and more complex systems will likely not scale very well. Employing the proper techniques for the given situation is essential to mitigate such concerns.

Chapter 2. Language tutorial

This tutorial introduces the CIF language. It explains the general idea behind the concepts of the language, and shows how to use them, all by means of examples. The tutorial is focused on giving a short introduction to CIF, and does not cover all details. It is recommended reading for all CIF users.

2.1. Introduction

CIF is primarily used to create models of physical systems and their controllers, describing their behavior. However, CIF is a general-purpose modeling language, and can be used to model practically anything, ranging from physical real-world systems to abstract mathematical entities.

CIF supports discrete event models, that are mostly concerned with what happens, and in which order. CIF also supports timed systems, where timing plays an explicit role, and hybrid systems, which combine the discrete events with timing. This makes CIF suitable for modeling of all kinds of systems.

The CIF tooling puts a particular focus on supporting the entire development process of controllers. However, just as the CIF language, the CIF tooling can be applied much more generally. The tooling allows among others specification, supervisory controller synthesis, simulation-based validation and visualization, verification, real-time testing, and code generation.

CIF originally stood for **C**ompositional **I**nterchange **F**ormat for hybrid systems. As the language has since evolved beyond its original purpose, the name 'CIF' is nowadays only used in its abbreviated form.

2.2. Lessons

Several lessons are available, grouped into the following categories:

- Basics
- Data
- Types and values
- Scalable solutions and reuse (1/2)
- Time
- Channel communication
- Functions
- Scalable solutions and reuse (2/2)
- Stochastics
- SVG visualization and interaction
- Text

- Synthesis
- Annotations

The lessons introduce new concepts, one by one, and are meant to be read in the given order.

Basics

Automata

Explains automata, locations, events, edges, transitions, and more.

Synchronizing events

Explains event synchronization, enabledness, traces, and state spaces.

Non-determinism

Explains multiple causes of non-determinism.

Alphabet

Explains alphabets for both individual automata and entire specifications.

Event declaration placement

Explains the placement of event declarations.

Shorter notations

Explains several shorter notations, including self loops, declaring multiple events with a single declaration, multiple events on an edge, and nameless locations.

Data

Discrete variables

Explains discrete variables, guards, and updates.

Discrete variable value changes

Explains how and when discrete variables can change value.

Location/variable duality (1/2)

Explains the duality between locations and variables using a model of a counter.

Location/variable duality (2/2)

Explains the duality between locations and variables using a model of a lamp.

Global read, local write

Explains the concepts of global read and local write.

Monitoring

Explains monitoring, self loops, and monitor automata.

Old and new values in assignments

Explains old and new values of variables in assignments, multiple assignments, and the order of assignments.

The `tau` event

Explains the `tau` event.

Initial values of discrete variables

Explains initialization of discrete variables, including the use of default values and multiple potential initial values.

Initialization predicates

Explains initialization in general, and initialization predicates in particular.

Using locations as variables

Explains the use of locations as variables.

State (exclusion) invariants

Explains state (exclusion) invariants.

State/event exclusion invariants

Explains state/event exclusion invariants.

Types and values

Types, values, and expressions

Explains the concepts of types, values, and expressions, as an introduction for the other lessons in this category.

Values overview

Provides an overview of the available values, and divides them into categories.

Integers

Explains integer types, values, and commonly used expressions.

Ranged integers

Explains ranged integers.

Reals

Explains real types, values, and commonly used expressions.

Booleans

Explains boolean types, values, and commonly used expressions.

Strings

Explains string types, values, and commonly used expressions.

Enumerations

Explains enumeration types, values, and commonly used expressions.

Tuples

Explains tuple types, values, and commonly used expressions.

Lists

Explains list types, values, and commonly used expressions.

Bounded lists and arrays

Explains bounded lists, arrays, and their relations with regular lists.

Sets

Explains set types, values, and commonly used expressions.

Dictionaries

Explains dictionary types, values, and commonly used expressions.

Combining values

Explains how to combine values of different types.

If and switch expressions

Explains `if` and `switch` expressions.

Scalable solutions and reuse (1/2)

Constants

Explains the use of constants.

Algebraic variables

Explains the use of algebraic variables.

Algebraic variables and equations

Explains the use of equations to specify values of algebraic variables.

Type declarations

Explains the use of type declarations.

Time

Timing

Introduces the concept of timing.

Continuous variables

Explains the use of continuous variables.

Continuous variables and equations

Explains the use of equations to specify values of continuous variables.

Equations

Show the use of equations for both continuous and algebraic variables, by means of an example of a non-linear system.

Variables overview

Provides an overview of the different kinds of variables in CIF, and their main differences.

Urgency

Explains the concept of urgency, as well as the different forms of urgency.

Deadlock and livelock

Explains the concepts of deadlock and livelock.

Channel communication

Channels

Explains point-to-point channels and data communication.

Dataless channels

Explains `void` channels that do not communicate any data.

Combining channel communication with event synchronization

Explains how channel communication can be combined with event synchronization, further restricting the communication.

Functions

Functions

Introduces functions, and explains the different kind of functions.

Internal user-defined functions

Explains internal user-defined functions.

Function statements

Explains the different statements that can be used in internal user-defined functions.

Functions as values

Explains using functions as values, allowing functions to be passed around.

Scalable solutions and reuse (2/2)

Automaton definition/instantiation

Explains using automaton definition and instantiation for reuse.

Parametrized automaton definitions

Explains parametrized automaton definitions.

Automaton definition parameters

Explains the different kinds of parameters of automaton definitions.

Groups

Explains hierarchical structuring using groups.

Group definitions

Explains groups definitions and parametrized group definitions.

Imports

Explains splitting CIF specifications over multiple files using imports.

Imports and libraries

Explains how to create libraries that can be used by multiple CIF specifications using imports, as well as how to use imports to include CIF specifications from other directories.

Imports and groups

Explains how imports and groups interact.

Namespaces

Explains namespaces, and how they can be used together with imports.

Input variables

Explains input variables, how they can be used for coupling with other models and systems, and their relation to imports.

Stochastics

Stochastics

Introduction to stochastic distributions, which allow for sampling, making it possible to produce random values.

Discrete, continuous, and constant distributions

Explains the different categories of stochastic distributions: discrete, continuous, and constant distributions.

Pseudo-randomness

Explains how computers implement stochastics using pseudo-random number generators, and how this affects the use of stochastics in CIF.

SVG visualization and interaction

SVG visualization

Introduction to SVG visualization, which allows to connect an image to a CIF model and update the image based on the state of the model.

First example

A first simple example of SVG visualization.

Inkscape

Explains how to use the Inkscape drawing program to edit SVG images.

Sun/moon example

An example of using groups of SVG objects, as well as controlling the visibility of objects based

on the current locations of automata.

Walk example

An example of using groups of SVG objects, positioning them, and changing their position using an output mapping, as well as using the `scale` standard library function, text formatting, multiple values in an output mapping, and multiple format specifiers in a format pattern.

Rate example

An example of using literal values in output mappings, as well using simple solution to solve visualization problems.

Workstation example

An example of fixed-value and clock-based object widths, changing the fill color of objects, updating multiple attributes of an SVG object, specifying colors in SVG, and improving scalability by reusing mappings.

Tank example

An example of using the `scale` standard library function, as well as how inverting the y-axis can reduce the number of output mappings.

Lamps example

An example of hierarchical scalability of SVG images by reusing mappings, as well as copying elements and moving them to the desired position.

Buffers/products example

Another example of hierarchical scalability of SVG images by reusing mappings, as well as copying (copied) elements and moving them to the desired position.

SVG interaction

Explains how to extend a model to couple it to an image for interaction via a visualization.

Text

Print output

Introduces print output by means of print declarations, which allows to print custom text from the model at runtime, either to the console or to a file.

Print output examples

Explains print output by means of examples.

Text formatting

Explains text formatting by means of examples.

Synthesis

Supervisory controller synthesis

Introduces supervisory controller synthesis.

Automaton kinds

Explains how to model automata with different automaton kinds.

Invariant kinds

Explains how to model invariants with different invariant kinds.

Event controllability

Explains the concept of event controllability and how to model events with different controllabilities.

Marker predicates

Explains the concept of marking and how to model marker predicates.

Annotations

Annotations

Explains how to annotate elements of the specification with extra information.

Controller properties annotations

Explains how certain tools can add properties of controllers to the specification.

Documentation annotations

Explains how to add documentation to elements of a specification.

Partial specification annotations

Explains how a specification can be labeled a 'partial' specification, making the type checker forego certain checks for it that often lead to false positive warnings on libraries.

Reachability requirement annotations

Explains how to add reachability requirements to a specification.

State annotations

Explains how certain tools that generate state spaces may annotate locations of a state space automaton with state information.

2.3. Basics

2.3.1. Automata

CIF models consist of *components*. Each of the components represents the behavior of a part of the system. Components can be modeled as *automata*, which form the basis of CIF. The following CIF *specification*, or CIF *model*, shows a simple automaton:

```
automaton lamp:
  event turn_on, turn_off;
```

```
  location on:
    initial;
    edge turn_off goto off;

  location off:
    edge turn_on goto on;
end
```

The automaton is named `lamp`, and not surprisingly represents the (discrete) behavior of a lamp.

Automaton `lamp` declares two *events*, named `turn_on` and `turn_off`. Events model things that can happen in a system. They represent changes. For instance, the `turn_on` event indicates that the lamp is being turned on. It represents the change from the lamp being off to the lamp being on. The event declaration in the `lamp` automaton declares two events. The event declaration only indicates that these events exist, it does not yet indicate when they can happen, and what the result of them happening is.

All automata have one or more *locations*, which represent the mutually exclusive *states* of the automaton. The `lamp` automaton has two *locations*, named `on` and `off`. Automata have an *active* or *current* location. That is, for every automaton one of its location is the active location, and the automaton is said to be *in* that location. For instance, the `lamp` automaton is either in its `on` location, or in its `off` location.

Initially, the lamp is on, as indicated by the `initial` keyword in the `on` location. That is, the `on` location is the initial location of the `lamp` automaton. The initial location is the active location of the automaton, at the start of the system.

In each location, an automaton can have different behavior, specified using *edges*. An edge indicates how an automaton can change its state, by going from one location to another. Edges can be associated with events, that indicate what happened, and thus what caused the state change. In each location, only the behavior specified by its edges is possible, for that automaton. No other behavior is possible.

The `lamp` automaton has an edge with the `turn_off` event, in its `on` location, going to the `off` location. Whenever the lamp is on, the `lamp` automaton is in its `on` location. Whenever the lamp is turned off, the `turn_off` event happens. The edge with that event indicates what the result of that event is, for the `on` location. In this case the result is that the lamp will then be off, which is why the edge goes to the `off` location.

The `lamp` automaton can go from one location to another, as described by its edges. This is referred

to as 'performing a transition', 'taking a transition', or 'taking an edge'. The `lamp` automaton can keep performing *transitions*. The lamp can be turned on, off, on again, off again, etc. This can go on forever.

2.3.2. Synchronizing events

The power of events is that they synchronize. To illustrate this, consider the following CIF specification:

```
automaton producer:
  event produce, provide;

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end
```

The automaton represents a producer that produces products, to be consumed by a consumer. The `producer` automaton starts in its `producing` location, in which it produces a product. Once the product has been produced, indicated by the `produce` event, the automaton will be in its `idle` location, where it waits until it can `provide` the produced product to the consumer. Once it has provided the product to the consumer, it will once again be `producing` another product. Consider also the following continuation of the above specification:

```
automaton consumer:
  event consume;

  location idle:
    initial;
    edge producer.provide goto consuming;

  location consuming:
    edge consume goto idle;
end
```

This second automaton represents a consumer that consumes products. The `consumer` is initially `idle`, waiting for a product from the producer. Once the producer has provided a product, the consumer will be `consuming`. Once it has consumed the product, as indicated by the occurrence of the `consume` event, it will become `idle` again.

The specification has three events, the `produce` and `provide` events declared in the `producer` automaton, and the `consume` event declared in the `consumer` automaton. The `consumer` automaton, in its `idle` location, has an edge that refers to the `provide` event declared in the `producer` automaton. As such, that edge and the edge in the `idle` location of the `producer` automaton, refer to the same event.

Synchronization

Events that are used in multiple automata, must *synchronize*. That is, if one of those automata performs a transition for that event, the other automata must also participate by performing a transition for that same event. If one of the automata that uses the event can not perform a transition in its current location, none of the automata can perform a transition for that event.

Now, lets take a closer look at the behavior of the producer/consumer example. Initially, the `producer` automaton is in its `producing` location, and the `consumer` automaton is in its `idle` location. Since the `producer` is the only automaton that uses the `produce` event, and there is an (outgoing) edge in its current location for that `produce` event, the `producer` can go to its `idle` location by means of that event.

Both the `producer` and `consumer` use the `provide` event. The `producer` has no edge with that event in its `producing` location, while the `consumer` does have an edge for that event in its `idle` location. Since events must synchronize, and the `producer` can not participate, the event can not occur at this time. This is what we expect, as the `producer` has not yet produced a product, and can thus not yet `provide` it to the consumer. The `consumer` will have to remain `idle` until the `producer` has produced a product and is ready to `provide` it to the `consumer`.

The `producer` blocks the `provide` event in this case, and is said to *disable* the event. The event is not blocked by the `consumer`, and is thus said to be *enabled* in the `consumer` automaton. In the entire specification, the event is *disabled* as well, as it is disabled by at least one of the automata of the specification, and all automata must enable the event for it to become enabled in the specification.

The only behavior that is possible, is for the `producer` to `produce` a product, and go to its `idle` location. The `consumer` does not participate and remains in its `idle` location. Both automata are then in their `idle` location, and both have an edge that enables the `provide` event. As such, the `provide` event is enabled in the specification. As this is the only possible behavior, a transition for the `provide` event is performed. This results in the `producer` going back to its `producing` location, while at the same time the `consumer` goes to its `consuming` location.

In its `producing` location, the `producer` can `produce` a product. Furthermore, in its `consuming` location, the `consumer` can `consume` a product. Two transitions are possible, and CIF does not define which one will be performed. That is, either one can be performed. No assumptions should be made either way. In other words, both transitions represent valid behavior, as described by this specification. Since only one transition can be taken at a time, there are two possibilities. Either the `producer` starts to `produce` the product first, and the `consumer` starts to `consume` after that, or the other way around.

Traces and state spaces

Once both transitions have been taken, we are essentially in the same situation as we were after the `producer` produced a product the first time, as both automata will be in their `idle` locations again. The behavior of the specification then continues to repeat forever. However, for each repetition different choices in the order of production and consumption can be made.

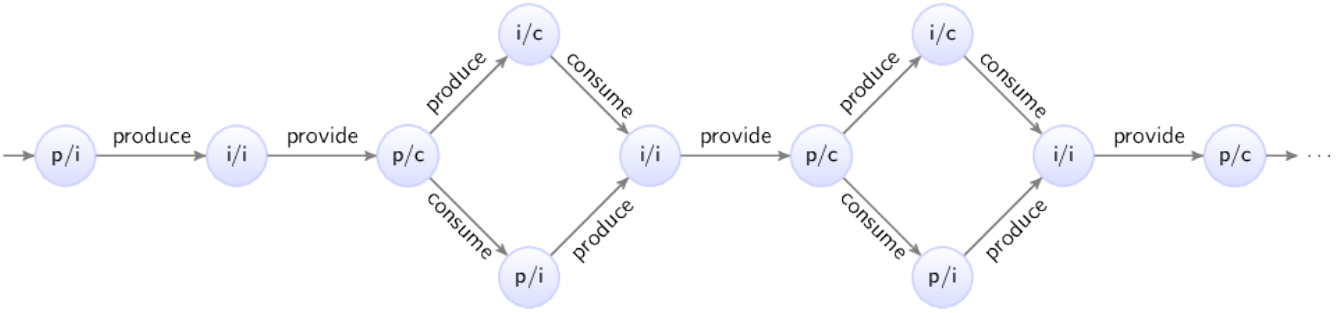
During a single *execution* or *simulation*, choices are made each time that multiple transitions are

possible. The sequence of transitions that are taken is called a *trace*. Examples of traces for the producer/consumer example are:

- `produce` → `provide` → `produce` → `consume` → `provide` → `produce` → `consume` → ...
- `produce` → `provide` → `produce` → `consume` → `provide` → `consume` → `produce` → ...
- `produce` → `provide` → `consume` → `produce` → `provide` → `produce` → `consume` → ...
- `produce` → `provide` → `consume` → `produce` → `provide` → `consume` → `produce` → ...

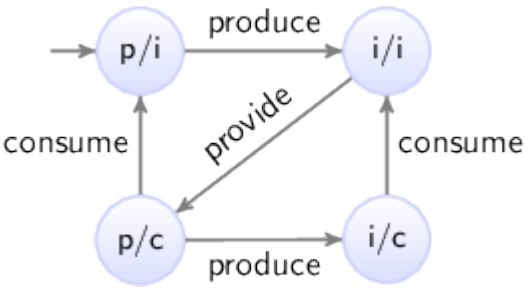
The traces end with `...` to indicate that they are partial traces, that go beyond the part of the trace that is shown. These four traces however, cover all the possibilities for the first seven transitions.

All possible traces together form the *state space*, which represents all the possible behavior of a system. For the producer/consumer example, the state space is:



Here the circles represent the states of the specification, which are a combination of the states of the two automata. The labels of the circles indicate the state, as a combination of the first letters of the locations of the automata. The initial state is labeled `p/i`, as initially automaton `producer` is in its `producing` (p) location, and the `consumer` is in its `idle` (i) location. The arrows indicate the transitions, and are labeled with events. The state space clearly shows the choices, as multiple outgoing arrows for a single state. It also makes it clear that as we move to the right, and make choices, we can make different choices for different products. Since the behavior keeps repeating itself, the state space ends with `...` to indicate that only a part of the state space is shown.

However, we can also show the entire behavior of the specification. Essential here is that the state space shown above has duplicate states. That is, several states have the same label, and allow for the same future behavior. By reusing states, a finite representation of the state space can be made, which shows the entire possible infinite behavior of the producer/consumer specification:



Concluding remarks

By using multiple automata, the producer and consumer were modeled independently, allowing for separation of concerns. This is an important concept, especially when modeling larger systems. In

general, the large system is decomposed into parts, usually corresponding to physical entities. Each of the parts of the system can then be modeled in isolation, with little regard for the other parts.

By using synchronizing events, the different automata that model different parts of a system, can interact. This allows for modeling of the connection between the different parts of the system, ensuring that together they represent the behavior of the entire system.

2.3.3. Non-determinism

Depending on the context in which it is used, non-determinism can mean different things. One definition is having multiple possible traces through the state space of a system. Another definition is having multiple possible transitions for a certain event, for a certain state. Different communities also use different definitions, and some communities only use one of the definitions, and use a different name to refer to the other concept.

Non-determinism between events

One cause of non-determinism is that multiple events are enabled, leading to multiple possible transitions. In other words, there are multiple possible traces through the state space. During the lesson on synchronizing events, we already encountered this form of non-determinism, as transitions for the `produce` and `consume` events could be performed in arbitrary order.

Non-determinism for single event

Another cause of non-determinism is the presence of multiple outgoing edges of a single location for the same event. This can lead to multiple possible transitions for a that event, for a single state. For instance, consider the following CIF specification:

```
automaton coin:
  event toss, land, pick_up;

  location hand:
    initial;
    edge toss goto air;

  location air:
    edge land goto ground;

  location ground:
    edge pick_up goto hand;
end

automaton outcome:
  location unknown:
    initial;
    edge coin.land goto heads; // First way to land.
    edge coin.land goto tails; // Second way to land.

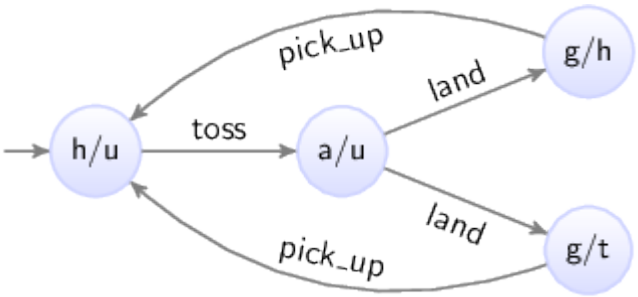
  location heads:
    edge coin.pick_up goto unknown;
```

```
  location tails:
    edge coin.pick_up goto unknown;
end
```

The `coin` automaton represents a single coin. Initially, it is in the `hand` of a person. That person can `toss` the coin up into the `air`, eventually causing it to fall and `land` on the `ground`. It can be picked up (event `pick_up`), causing it to once again be in the `hand` of a person.

The `outcome` automaton registers the outcome of the coin toss. Initially, the outcome is `unknown`. Whenever the coin is tossed, it lands (event `land` from automaton `coin`) on the ground with either the `heads` or `tails` side up. The `unknown` location of the `outcome` automaton has two edges for the same event. This leads to two possible transitions, one to the `heads` location, and one to the `tails` location. This is a non-deterministic choice, as the model does not specify which transition is chosen, or even which choice is more likely.

In both the `heads` and `tails` locations, the coin can be picked up again, making the outcome `unknown`. The coin can be tossed again and again, repeating the behavior forever. The following figure shows the state space of this specification:



The states are labeled with the first letters of the current locations of the two automata.

2.3.4. Alphabet

The lesson on synchronizing events described how events that are used in multiple automata exhibit synchronizing behavior. That is, if the event is used in multiple automata, they must all enable that event in order for a transition to be possible. If one of them can not perform the event, the event is disabled, and none of the automata can perform a transition for that event.

Whether an automaton participates in the synchronization for a certain event, is determined by its *alphabet*. The alphabet of an automaton is the collection of events over which it synchronizes.

Default and implicit alphabets

By default, the alphabet of an automaton implicitly contains all the events that occur on the edges of the automaton. For instance, consider the following CIF specification (the producer/consumer example from the lesson on synchronizing events):

```
automaton producer:
  event produce, provide;

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end

automaton consumer:
  event consume;

  location idle:
    initial;
    edge producer.provide goto consuming;

  location consuming:
    edge consume goto idle;
end
```

The alphabet of the `producer` automaton contains the events `produce` and `provide`, as both occur on edges of that automaton. The alphabet of the `consumer` automaton contains the events `producer.produce` and `consume`.

Explicit alphabet

It is possible to explicitly specify the alphabet of an automaton, as follows:

```
event provide;

automaton producer:
  event produce;

  alphabet produce, provide; // Alphabet explicitly specified.

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end
```

The `alphabet` keyword is followed by the events that comprise the alphabet of the automaton,

separated by commas. In this case, the alphabet contains the `produce` and `provide` events. Since this explicitly specified alphabet is exactly the same as the default alphabet, it could just as easily be omitted.

Non-default alphabet

The alphabet is allowed to be empty, which can be explicitly specified as follows:

```
alphabet; // Empty alphabet. Automaton doesn't synchronize over any events.
```

However, the alphabet of an automaton must at least contain the events that occur on the edges of an automaton. That is, it must at least contain the default alphabet.

It may however also contain additional events. Since there are no edges for those additional events, the automaton can never enable those events, and thus always disables them. If a single automaton disables an event, and since it must always participate if it has that event in its alphabet, this means that the event becomes globally disabled in the entire specification. Having such additional events in the alphabet leads to a warning, to inform about the potential undesired effects of globally disabling events in this manner.

Implicit vs explicit

It should be clear that for most automata, the implicit default alphabet suffices. There are however reasons for explicitly specifying the default alphabet. For large automata, it can improve the readability, as the explicit alphabet makes it easy to determine the alphabet of the automaton, without having to look at all the edges.

The need to explicitly specifying a non-default alphabet rarely occurs. However, several tools generate CIF specifications with explicit alphabets.

2.3.5. Event declaration placement

Consider the following CIF specification (the producer/consumer example from the lesson on synchronizing events):

```
automaton producer:
  event produce, provide;

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end
```

```
automaton consumer:
  event consume;

  location idle:
    initial;
    edge producer.provide goto consuming;

  location consuming:
    edge consume goto idle;
end
```

The specification could also be specified as follows:

```
automaton producer:
  event produce, provide, consume; // Declaration of event 'consume' moved.

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end

automaton consumer:
  location idle:
    initial;
    edge producer.provide goto consuming;

  location consuming:
    edge producer.consume goto idle; // Event 'consume' from 'producer'.
end
```

The `consume` event is now declared in the `producer` automaton rather than the `consumer` automaton, but the locations and edges have not changed. This modified specification exhibits the same behavior as the original.

It should be clear that while events can be declared in various places, it is best to declare them where they belong. That is, the `consume` event is only used by the `consumer` automaton, and is thus best declared in that automaton. Similarly, the `produce` event is only used by the `producer` automaton.

The `provide` event however is used by both automata. In such cases the event is usually declared where it is initiated. In the example above, the producer provides the product to the consumer, and not the other way around. Therefore, the `provide` event is declared in the `producer` automaton, rather than in the `consumer` automaton.

However, the modeler is free to choose the best place to declare the event. If no choice can be made

between the automata, the event can also be declared outside the automata, as follows:

```
event provide; // Event 'provide' now declared outside the automata.

automaton producer:
  event produce;

  location producing:
    initial;
    edge produce goto idle;

  location idle:
    edge provide goto producing;
end

automaton consumer:
  event consume;

  location idle:
    initial;
    edge provide goto consuming; // Can directly refer to 'provide' event.

  location consuming:
    edge consume goto idle;
end
```

This specification also has the same behavior. Only the placement of the event declarations has changed.

The place where an event is declared is of no influence to the implicit (default) alphabets of the automata. The implicit alphabet of an automaton is determined solely based on the events that occur on the edges of that automaton.

2.3.6. Shorter notations

This lessons explains several short notations, that can be used for easier modeling, can reduce the size of the specification, and make specifications easier to read. The following topics are discussed:

- Self loop
- Declaring multiple events with a single declaration
- Multiple events on an edge
- Nameless location

Self loop

A self loop is an edge that goes to the location from which it originated. Consider the following

example:

```
automaton a:
  event e;

  location x:
    edge e goto x;
end
```

The edge in location `x` of automaton `a` goes to location `x`. The effect of the `e` event is that automaton `a` remains in its `x` location. A self loop can be used to allow a certain event, essentially ignoring it for that location.

The following short notation can be used for self loops:

```
automaton a:
  event e;

  location x:
    edge e; // Goto omitted for self loop.
end
```

The `goto` part of the edge can be omitted for self loop. This can help make the model easier to read, as the resulting location does not have to be checked against the source location of the edge, to see whether they match.

Declaring multiple events with a single declaration

Several of the previous lessons already showed that multiple events can be declared using a single event declaration:

```
event a, b, c; // Single declaration declares multiple events.
```

This is equivalent to using multiple event declarations that each declare a single event:

```
event a; // Multiple declarations each declare a single event.
event b;
event c;
```

Using a single declaration to declare multiple events can help reduce the length of a specification.

Multiple events on an edge

Consider the following CIF specification:

```
automaton a:
  event e, f;

  location loc:
    edge e goto loc; // Two edges that only differ in the event.
    edge f goto loc;
end
```

This can also be written more compactly, as follows:

```
automaton a:
  event e, f;

  location loc:
    edge e, f goto loc; // Edge with two events.
end
```

An edge with two or more events means exactly the same as having individual edges for the different events. That is, a choice is made between them. It does *not* specify that event `f` can only happen after event `e` has already happened. Both automata thus have the exact same behavior.

Nameless location

Several of the examples above show automata with a single location. For such automata, the name of the location is optional:

```
automaton a:
  event e;

  location:
    edge e;
end
```

Since the location has no name, there is no way to refer to it in a `goto`, and thus only self loop edges can be used. Leaving out the name prevents having to come up with a dummy name, which can clutter the specification.

2.4. Data

2.4.1. Discrete variables

This lesson introduces discrete variables. Consider the following specification:

```
automaton counter:
  event increment, decrement;

  disc int count = 3;

  location:
    edge decrement when count > 0 do count := count - 1;
    edge increment when count < 5 do count := count + 1;
end
```

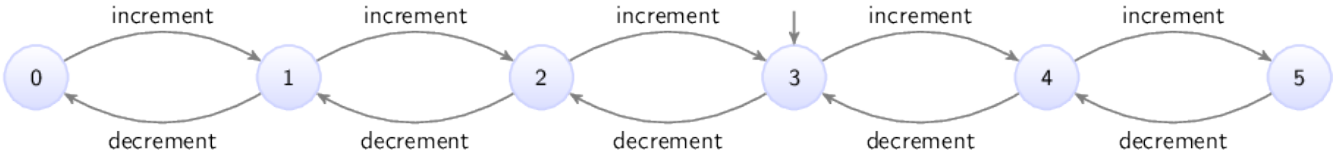
The `counter` automaton can be used to count certain things. The `increment` and `decrement` events are used to change the count. The count itself is stored in a variable named `count`. CIF has several different types of variables. Here, we use a discrete variable, as indicated by the `disc` keyword. The variable has an `int` data type, meaning it can have integer numbers as its value. It is initialized to value `3`.

The automaton has two edges, one for the `increment` event, and one for the `decrement` event. The edge for the `decrement` event has a *guard* that indicates under which circumstances the event can take place. The condition is indicated using the `when` keyword. In this case, the guard ensures that the count can only be decremented if it is currently positive. The guard of the edge for the `increment` event indicates that the count can only be incremented as long as it is less than five. In general, a guard must hold in the source location of the edge, for the edge to be enabled, and a transition to be possible. If the guard is not specified, it defaults to `true`, which always holds and does not restrict the edge in any way.

Both edges also have *updates*, indicated using the `do` keyword. Updates can be used to specify the effect of the transition on variables. In this case, the updates *assign* a new value to the `count` variable that is one less or one more than the current value. That is the value of the `count` variable is decremented or incremented by one.

This specification represents a counter that can be repeatedly incremented and decremented by one, and ensures that the value of variable `count` is always at least zero and at most five.

The state space of the `counter` automaton is as follows:



2.4.2. Discrete variable value changes

Discrete variables can only change value by explicitly assigning them a new value in the `do` part of an edge. If an edge does not assign a value to a discrete variable, that variable keeps its current value. Consider the following CIF specification:

```
automaton lamp:
  event turn_on, turn_off;

  disc int count = 0;

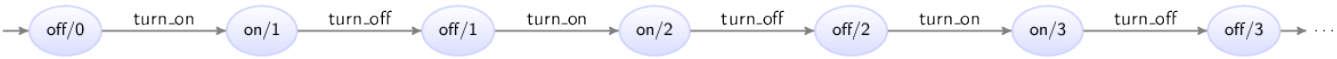
  location off:
    initial;
    edge turn_on  do count := count + 1 goto on;

  location on:
    edge turn_off                       goto off;
end
```

This is the same `lamp` automaton as used in the lesson on automata, but with a `count` variable added. This variable is used to count the number of times that the lamp has been turned on. The edge for the `turn_on` event increments the value of the variable by one, each time the lamp is turned on.

The edge for the `turn_off` event does not assign a value to a variable, so variable `count` keeps its value when the lamp is turned off.

The state space of this specification is:



The states are labeled with the name of the current location of automaton `lamp` and the current value of variable `count`.

2.4.3. Location/variable duality (1/2)

The lesson that introduces discrete variables, uses an example of a counter. The actual count was modeled using a variable:

```
automaton counter:
  event increment, decrement;

  disc int count = 3;

  location:
    edge decrement when count > 0 do count := count - 1;
    edge increment when count < 5 do count := count + 1;
end
```

It is also possible to use multiple locations instead of a variable:

```
automaton counter:
  event increment, decrement;

  location zero:
```

```
    edge increment goto one;

  location one:
    edge decrement goto zero;
    edge increment goto two;

  location two:
    edge decrement goto one;
    edge increment goto three;

  location three:
    initial;

    edge decrement goto two;
    edge increment goto four;

  location four:
    edge decrement goto three;
    edge increment goto five;

  location five:
    edge decrement goto four;
end
```

This alternate model has the same behavior, in that it models a counter that can be incremented an decremented in steps of one, and the value is kept at least zero, and at most five. The variant with the variable however, is shorter and more intuitive. It is also easier to change to the `count < 5` guard to `count < 100` than it is to add dozens of additional locations and edges. In this case, using a variable is preferable to using multiple locations.

2.4.4. Location/variable duality (2/2)

The lesson that introduces automata, used an example of a lamp:

```
automaton lamp:
  event turn_on, turn_off;

  location on:
    initial;
    edge turn_off goto off;

  location off:
    edge turn_on goto on;
end
```

The automaton uses two locations to keep track of the current state of the lamp. Instead of two locations, it is also possible to use a variable:

```
automaton lamp:
  event turn_on, turn_off;

  disc bool on = true;

  location:
    initial;
    edge turn_on  when not on do on := true;
    edge turn_off when     on do on := false;
end
```

This alternate automaton uses a single variable named `on`. The data type of the variable is `bool`, which means that the variable can only have one of two possible values: `true` or `false`. If variable `on` has value `true`, the lamp is on, and if it has value `false` it is off. Initially, the lamp is on, as the initial value of the variable is `true`. The automaton has only one location, with two edges. The first edge indicates that the lamp can be turned on (event `turn_on`), only when it is not currently on (guard `not on`), and then afterwards is on (variable `on` becomes `true`). Similarly, the second edge indicates that the lamp can be turned off, only when it is currently on, and then afterwards is on.

Both models represent a lamp that is initially on, and can be turned off, on, off again, on again, etc, repeating the behavior forever. Which approach is best depends on your preference, and on the rest of the model. It is however also possible to use both locations and a variable:

```
automaton lamp:
  event turn_on, turn_off;

  disc bool on2 = true;

  location on:
    initial;
    edge turn_off do on2 := false goto off;

  location off:
    edge turn_on  do on2 := true  goto on;
end
```

This automaton has the same behavior as the previous two automata. Variable `on` is renamed to `on2`, as a variable can not have the same name as a location of that same automaton.

While it is possible to model a lamp like this, this automaton duplicates the information about whether the lamp is on or off. This makes the automaton larger and more complex than it needs to be. In general, it is usually better to choose either a variable, or multiple locations, to express something, and not both. In several future lessons, we'll see that combining multiple locations with variables is useful, but not to express the same thing. Furthermore, an other future lesson explains how to use a location as a variable.

2.4.5. Global read, local write

Discrete variables can only be declared in automata, and may only be assigned (given a value, written) by that automaton. They may however be read globally. Consider the following CIF specification:

```
automaton customer:
  location:
    initial;
    edge queue1.enter when queue1.count <= queue2.count;
    edge queue2.enter when queue2.count <= queue1.count;
end

automaton queue1:
  event enter, leave;

  disc int count = 0;

  location:
    initial;
    edge enter when count < 2 do count := count + 1;
    edge leave when count > 0 do count := count - 1;
end

automaton queue2:
  event enter, leave;

  disc int count = 0;

  location:
    initial;
    edge enter when count < 2 do count := count + 1;
    edge leave when count > 0 do count := count - 1;
end
```

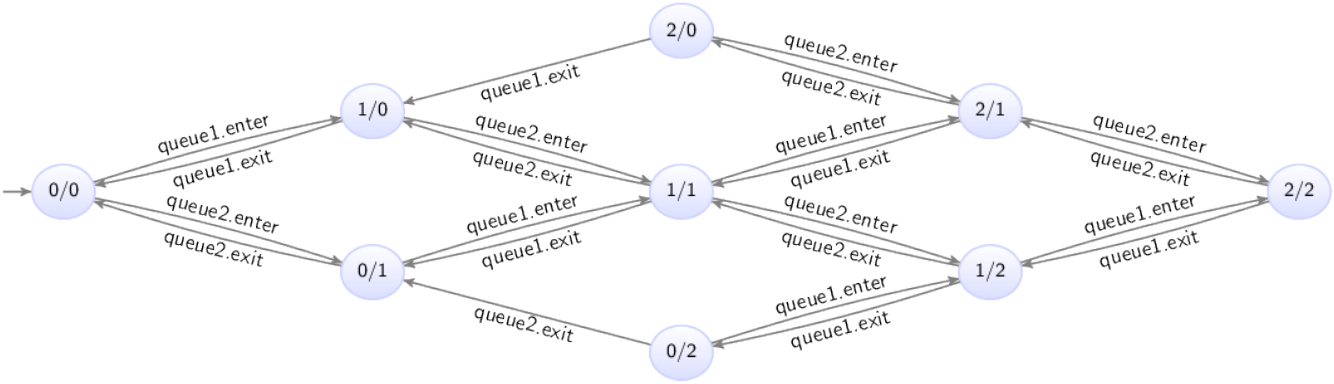
This specification models a supermarket, and features a customer and two queues. Customers arrive and enter either of the queues. Eventually customers leave the queue.

Both queues (automata `queue1` and `queue2`) are identical, except for their names. They maintain the `count`, which represents the number of customers in the queue. A queues is full if it contains two customers. Customers can thus only `enter` a queue if less than two customers are present. Similarly, it is only possible for a customer to leave a queue if there is a customer in the queue.

Customers decide to which queue they go, based on the number of customers already present in those queues. A customer only enters the first queue if that queue has less than or the same number of customers as the second queue. Similarly, a customer only enters the second queue if that queue has less than or the same number of customers as the first queue. If the queues have the same number of customers, the customer can choose either queue.

The `enter` event declared in the first queue (`queue1`) is used by both the `customer` automaton and the `queue1` automaton. The event is thus only possible (enabled) if both automata can participate. Both automata restrict the occurrence of the event using a guard. The event is thus only possible if both guards hold. That is, a custom never enters the first queue if it is full, but it also never enters that queue if it has more customers than the second queue.

The state space of this specification is as follows:



The states are labeled with the counts of the first and second queues.

The `customer` automaton uses the values of the variables of the queue automata, and thus reads variables of other automata. This is allowed, due to the *global read* concept of CIF. This concept allows for short guards, that directly and intuitively represent the condition under which an event may take place.

The global read concept should only be used when it is intuitive. In the supermarket example, the customer can physically see how many customers are in the queues. It is therefore intuitive to directly refer to the `count` variables of the queue automata. If however the customer would not be able to physically observer the queues, then the customer would not be able to directly base its decision of which queue to join, on that information. In that latter case, it may not be a good idea to model the guard in such way.

The *local write* concept means that discrete variables can only be written by the automata in which they are declared. It is not allowed for the `customer` and `queue2` automata to write (change the value of) the `count` variable of the `queue1` automaton. Only the `queue1` automaton may write that variable. The local write concept prevents that multiple automata write to the same variable, as part of a synchronizing event, potentially causing conflicting values to be assigned to that variable. This leads to several benefits, most notably simpler semantics.

2.4.6. Monitoring

This lesson explains the concept of *monitoring*. It is explained using the following CIF specification:

```
automaton producer:
  event produce, provide_a, provide_b;

  location producing:
    initial;
    edge produce goto idle;
```

```
  location idle:
    edge provide_a goto producing;
    edge provide_b goto producing;
end

automaton detect_changeover:
  disc int count = 0;

  location start:
    initial;
    edge producer.provide_a goto a;
    edge producer.provide_b goto b;

  location a:
    edge producer.provide_b do count := count + 1 goto b;

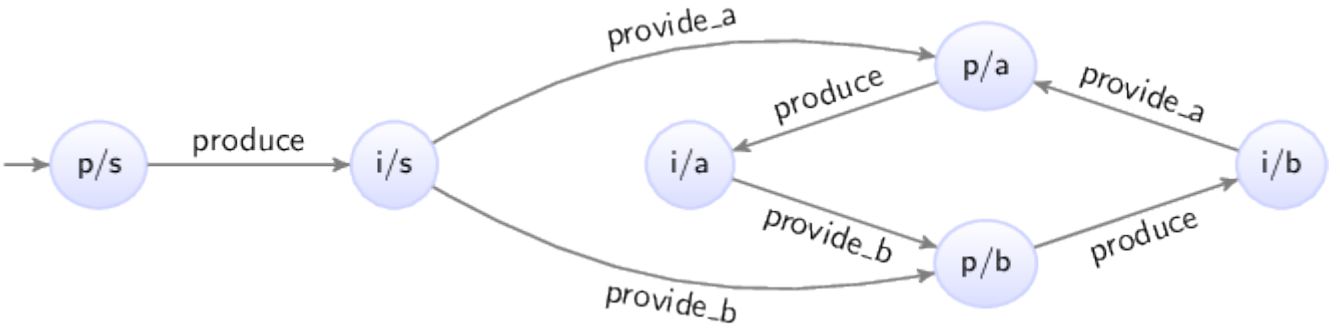
  location b:
    edge producer.provide_a do count := count + 1 goto a;
end
```

The `producer` automaton represents a producer that can repeatedly `produce` a product, and provide it to either consumer 'a' (event `provide_a`) or consumer 'b' (event `provide_b`). The consumers are not modeled.

The `detect_changeover` automaton detects consumer changes. That is, it detects and counts how often the producer switching from providing consumer 'a' with products to providing consumer 'b' with products, and vice versa. Initially, the automaton waits for the first product to be provided. It goes to either location `a` or location `b`, depending on which consumer is provided that first product. Whenever a product is then provided to the other consumer, the `count` is incremented by one to account for the changeover taking place. This also switches the location to the location for the other consumer, where once again the automaton waits for a changeover.

The monitoring problem

There is a problem with the `detect_changeover` automaton. In its `a` location, it disables the `provide_a` event, as there is no edge for that event, and the automaton has that event in its (implicit) alphabet. This means that after a product is provided to consumer 'a', no more products can be provided to that same consumer, until the producer provides a product to the consumer 'b', and the automaton switches to the corresponding `b` location. However, the idea is that the producer can provide products to either consumer, at all times, as that is the way it is intended. The `detect_changeover` automaton currently prevents behavior that is present in the `producer`, while it is only meant to *observe* or *monitor* products being provided. The state space of the specification is:



The states are labeled with the first letters of the names of the current locations of the automata. Note how the `i/a` and `i/b` locations only have outgoing transitions for either the `provide_a` transition or the `provide_b` transition.

Monitoring with self loops

A simple solution is to allow the disabled events:

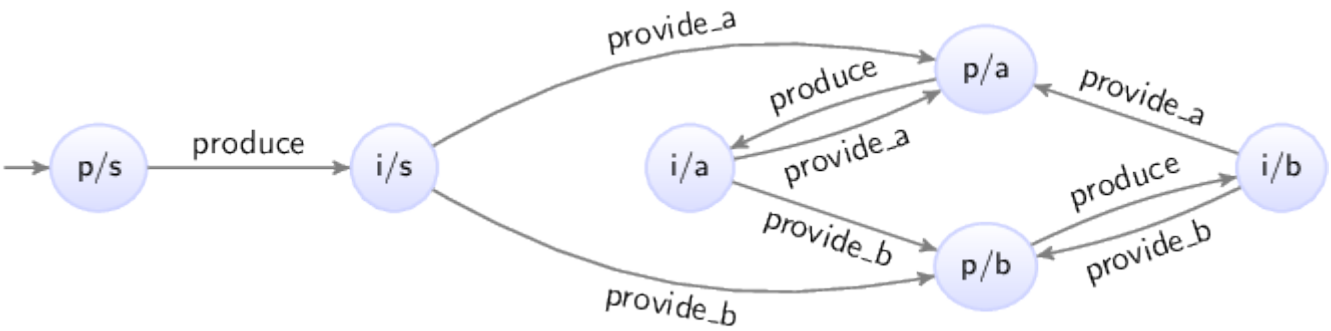
```
automaton detect_changeover:
  disc int count = 0;

  location start:
    initial;
    edge producer.provide_a goto a;
    edge producer.provide_b goto b;

  location a:
    edge producer.provide_a; // Added self loop.
    edge producer.provide_b do count := count + 1 goto b;

  location b:
    edge producer.provide_a do count := count + 1 goto a;
    edge producer.provide_b; // Added self loop.
end
```

The `provide_a` event has been added to an edge in the `a` location. The edge is a self loop, meaning the current location of automaton `detect_changeover` does not change as a result of taking the edge. This means that essentially the event is ignored by the `detect_changeover` automaton, as the edge also has no updates. The state space of the modified specification is:



Now, whenever the `provide_a` event is possible, the `provide_b` event is also possible, and vice versa, just as in the `producer` automaton. The `detect_changeover` automaton no longer restricts the occurrence of the events; it only monitors them.

Monitoring with monitor automata

An alternative to adding self loops, is to use a *monitor* automaton. A monitor automaton is an automaton that monitors or observes one or more events. The events that it monitors, are never blocked (disabled) by that automaton. For our producer/changeover example, we can turn the `detect_changeover` automaton into a monitor automaton for the `provide_a` and `provide_b` events:

```
automaton detect_changeover:
  monitor producer.provide_a, producer.provide_b; // Monitor instead of the self
loops.

  disc int count = 0;

  location start:
    initial;
    edge producer.provide_a goto a;
    edge producer.provide_b goto b;

  location a:
    edge producer.provide_b do count := count + 1 goto b;

  location b:
    edge producer.provide_a do count := count + 1 goto a;
end
```

By default, automata don't monitor any events. Using a `monitor` declaration with one or more events, turns the automaton into a monitor automaton for those events. For the producer/changeover example, the behavior with the monitor automaton is exactly identical to the behavior of the specification with the self loops.

By omitting the events from the `monitor` declaration, an automaton monitors all events of its alphabet:

```
monitor; // Monitor all events in the alphabet of the automaton.
```

For the producer/changeover, which has only the `provide_a` and `provide_b` events in its alphabet, this would result in the same behavior as for the automaton that monitors the two events explicitly.

Using a monitor automaton instead of self loops has several advantages. A monitor declaration has to be provided only once, while self loops often have to be added to several locations. Furthermore, if the automaton is changed, it may be necessary to add or remove self loops, while the monitor declaration can most often be kept as is.

2.4.7. Old and new values in assignments

This lesson explains old and new values of variables in assignments, multiple assignments, and the order of assignments.

Old and new values

Consider the following CIF specification:

```
automaton counter:
  event increment;

  disc int count = 0;

  location:
    initial;
    edge increment do count := count + 1;
end
```

The `counter` automaton represents a counter that starts counting at zero, and can be incremented in steps of one.

In assignments, the part to the left of the `:=` is called the *left hand side* of the assignment, or the *addressable*. The addressable is the variable that is assigned, and gets the new value. In the example above, variable `count` is assigned a new value.

The part to the right of the `:=` is called the *right hand side* of the assignment, or the (new) *value*. In the example above, the new value is computed by taking the *current* or *old* value of variable `count` and incrementing it by one.

In general, for variables used to compute the new value, always the old value of those variables are used. The new values for variables after a transition, are always computed from the old values of variables from before that transition.

Multiple assignments

It is allowed to update multiple variables on a single edge, leading to multiple variables getting a new value as part of a single transition. For instance, consider the following CIF specification:

```
automaton swapper:
  event swap;
  disc int x = 0, y = 0;

  location:
    initial;
    edge swap do x := y, y := x + 1;
```

```
end
```

The `swapper` automaton declares two variables, `x` and `y`. It keeps swapping the values of both variables, each time increasing the value of `y` by one.

Initially, both variables have value zero. During the first `swap`, variable `x` gets the value of variable `y`. Since the old values of the variables are used to compute the new values, variable `x` remains zero. Variable `y` gets the old value of variable `x`, which is also zero, incremented by one. The result of the first swap is that `x` remains zero and `y` becomes one.

During the second swap, `x` gets the value of variable `y`, which is then one. Variable `y` gets the value of variable `x`, which was still zero before the second swap, incremented by one. Both variables are thus one after the second swap.

During the third swap, `x` gets the value of variable `y` from after the second swap, and thus remains one. Variable `y` becomes two.

The state space of this somewhat artificial example is as follows:



The states are labeled with the values of variables `x` and `y`.

Assignment order

It is important to note that since the new values of the variables are computed from the old values of the variables, assignments are completely independent of each other. In the example above, variable `x` is assigned a new value in the first assignment, and variable `x` is also used to compute the new value of variable `y`. However, the old value of variable `x` is used to compute the new value of variable `y`. Therefore, the assignment to `x`, which indicates how `x` should be given a new value, has no effect on the new value off `y`, as the old value of `x` is used for that, regardless of whether `x` is assigned a new value.

Since assignments are independent of each other, the order of the assignments of the edge does not matter. Consider the following alternative edge:

```
edge swap do y := x + 1, x := y;
```

The assignments to `x` and `y` have been reordered. The behavior of the specification does not change as a result of this reordering.

Multi-assignments

CIF supports both multiple assignments as well as multi-assignments. To see the difference,

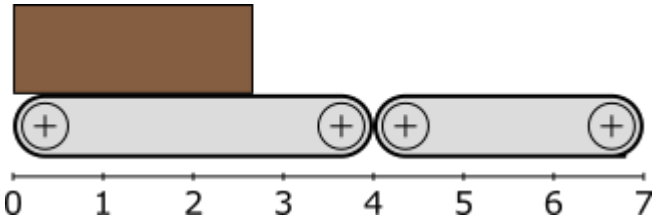
consider the following examples:

```
edge ... do x := y, y := x + 1;     // Multiple (two) assignments.
edge ... do (x, y) := (y, x + 1);   // Single multi-assignment.
```

The first edge has multiple assignments, namely one assignment to variable `x` and one assignment to variable `y`. The second edge has one assignment, that gives new values to variables `x` and `y`. Both are identical, in that they have the same affect: variable `x` is given the old value of variable `y` and variable `y` is given the old value of variable `x` incremented by one. Generally, using multiple assignments is preferred over using multi-assignments, as the former is easier to read. However, in certain cases, such as for tuple unpacking, only the latter variant can be used.

Event synchronization and assignment order

Consider a system with two conveyors. Products enter on the first conveyor, and move towards the second conveyor. Once they leave the first conveyor, they move onto the second one. Once they exit from the second conveyor, they leave the system. The positions of the left sides of the boxes are in range zero to seven, as indicated in the following figure:



This system can be modeled using the following CIF specification:

```
event move;

automaton conveyor1:
  monitor move;
  event exit1;
  disc int pos = 0;

  location:
    initial;
    edge move  when pos < 4 do pos := pos + 1;
    edge exit1 when pos = 4 do pos := 0;
end

automaton conveyor2:
  monitor move;
  event exit2;
  disc int pos = -1;

  location:
    initial;
    edge conveyor1.exit1 when pos = -1             do pos := conveyor1.pos;
    edge move            when pos >= 0 and pos < 7 do pos := pos + 1;
```

```
    edge exit2          when pos = 7            do pos := -1;
end
```

Each conveyor is modeled using an automaton. Both conveyors use a `pos` variable to represent the position of the left side of the box. The first conveyor gets a new box as soon as one leaves. The second one has to wait for a box from the first, and can thus be without a box. This is represented by value `-1` for the `pos` variable from automaton `conveyor2`. The `-1` value is not a actual position, but a special value indicating that no box is present on the conveyor.

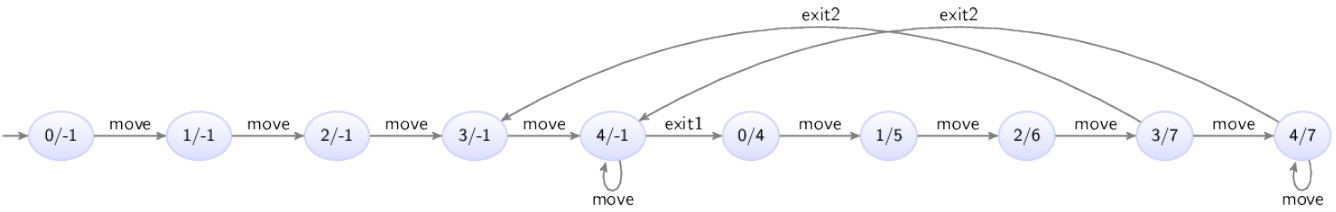
Boxes on the first conveyor can move towards the second conveyor (event `move`), until they reach position 4. They then leave the first conveyor (event `exit1`), and a new box immediately enters the first conveyor (variable `pos` is reset to zero).

Boxes enter the second conveyor when they leave the first conveyor (event `exit1` from `conveyor1`). The position of the box is then transferred from the first conveyor to the second. The box keeps moving (event `move`) on the second conveyor until it reaches position 7. At position 7 it leaves (event `exit2`) the second conveyor, and the system.

Both automata synchronize over the `move` event, meaning that the boxes on both conveyors move at the same time. Both automata monitor that event to ensure it is never blocked if only the other conveyor can actually move.

Both automata synchronize over the `exit1` event. The first conveyor resets is own position (variable `pos`) to zero. The second conveyor sets its own position (variable `pos`) to the position of the first conveyor. Since old values of variables are used to compute the new values, the new value of variable `pos` in `conveyor2` is given the old value of variable `pos` from `conveyor1`. This is not influenced by the assignment to variable `pos` of `conveyor1` to zero, as assignments are independent, and the order of assignments does not matter, just as for multiple assignments on a single edge.

The state space of this specification is as follows:



The states are labeled with the values of the `pos` variables of the automata for the first and second conveyors.

The important part of the state space is the transition from state `4/-1`, where the box of the first conveyor is at the end and the second conveyor has no box, to state `0/4`, where the first conveyor has received a new box at position zero, and the second conveyor has taken over the box (and the administration of its position) from the first conveyor.

2.4.8. The `tau` event

Events allow for synchronization, allowing for interaction between automata based on events. If however an automaton has an edge that performs some internal processing, the event may not always be relevant. Consider for instance the following CIF specification:

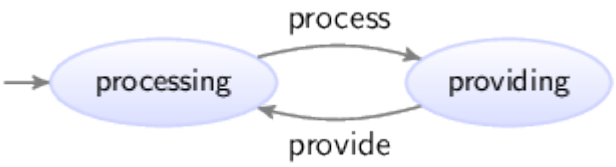
```
automaton machine1:
  event process, provide;
  disc int id = 0;

  location processing:
    initial;
    edge process do id := id + 1 goto providing;

  location providing:
    edge provide goto processing;
end

automaton machine2:
  location:
    initial;
    edge machine1.provide;
end
```

The specification models two machines. Products enter the first machine, which processes them (event `process`) and assigns them an `id`. The machine them provides (event `provide`) them to the second machine. The second machine currently just accepts the products provided by the first machine, but would in reality likely perform its own processing as well. The state space of the specification is as follows:



The states are labeled with the names of the current locations of automaton `machine1`. Since automaton `machine2` has only a single location, its current location does not change, and it is therefore not included in the state names.

The `provide` event synchronizes over both automata, while the `process` event is local to the first machine. The `process` event is not essential, and could be left out:

```
automaton machine1:
  event provide;                          // No more 'process' event.
  disc int id = 0;

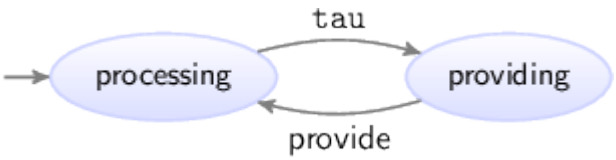
  location processing:
    initial;
    edge do id := id + 1 goto providing; // No more event on the edge.

  location providing:
    edge provide goto processing;
end

automaton machine2:
```

```
  location:
    initial;
    edge machine1.provide;
end
```

By omitting the event from an edge, the `tau` is used for that edge. The `tau` event is an event that is implicitly always present without declaring it. The state space of this modified specification is:



The `tau` event does not synchronize. You can think of this as each automaton having its own local `tau` event, and since then they are different events, they do not synchronize. If multiple automata can perform a transition for an edge with the `tau` event, this leads to potential transitions for each of those edges. Since they are all labeled with the `tau` event, it is impossible to distinguish them solely based on their label. This is a form of non-determinism.

Using the `tau` events saves having to declare a local event, and also saves having to put that event on the edge. It thus leads to smaller specifications. However, as explained above, if `tau` is used on multiple edges of multiple automata, the different `tau` transitions can no longer be distinguished from each other in the state space. The use of the `tau` event is thus always a trade-off.

It is also possible to explicitly use the `tau` event:

```
edge tau goto ...;
```

The `tau` event can thus be used instead of 'regular' events, and may even be combined with 'regular' events on the same edge:

```
edge provide, tau goto ...;
```

Omitting the events from an edge defaults to a single `tau` event, as shown in one of the examples above.

2.4.9. Initial values of discrete variables

Discrete variables can be given an initial value with their declaration:

```
disc int x = 1;
```

The initial value may be omitted, leading to the default value of its data type being used:

```
disc int x;
```

```
disc bool y;
```

The default value of integer typed variables is `0`. The default value of boolean typed variables is `false`.

It is possible to indicate that a variable has more than one potential initial value:

```
disc int x in {1, 2, 4};
```

This declares a variable `x` that has three potential initial values. Variables can only have one value at a time, so an initial value has to be chosen from the set of potential initial values. In other words, initially the value of variable `x` is either `1`, `2`, or `4`. For information on how to store multiple values in a single variable, see the lessons on types and values, in particular those on tuples, lists, and sets.

It is also possible to indicate that a variable can have any arbitrary initial value:

```
disc int x in any;
disc bool y in any;
```

Variable `x` can initially have any value. The only constraint is that the initial value must be an integer value, as it must conform to the integer type (`int`) of the variable. Examples of initial values include `-1027`, `0`, `1`, and `12345`. Variable `y` can initially have any value, as long as that value is a boolean value, due to the variable having a boolean type (`bool`). There are only two boolean values, `true` and `false`.

Discrete variables with multiple potential initial values and arbitrary initial values essentially *parametrize* the specification. The exact initial value is to be chosen or *configured* later on. This allows a single specification to be used for various different combinations of initial values.

So far all examples used literal values to initialize the variables. However, it is also allowed to use expressions to compute initial values, for instance based on the initial values of other variables:

```
disc int x = 1;     // Initial value: 1
disc int y = x * 2; // Initial value: 2
disc int z = x + y; // Initial value: 3
```

Variable `x` is explicitly initialized with value `1`. Variable `y` is initialized to the initial value of `x`, multiplied by two. Variable `z` is initialized to the sum of the initial values of `x` and `y`. Using this kind of initialization is useful if the initial values must be kept consistent. Changing the initial value of `x` automatically also changes the initial values of `y` and `z`.

The order of the declaration of the variables does not matter. We could just as easily declare them

as follows:

```
disc int y = x * 2; // Uses variable x, which is declared later.
disc int x = 1;
```

Variable y is still initialized using the initial value of variable x, which is now declared after variable y. It is not allowed to construct loops, where the initial values of variables depend on each other:

```
disc int x = y; // Invalid initial value due to cyclic dependency.
disc int y = z;
disc int z = x;
```

Variable x uses the value of variable y, which uses the value of variable z, which in turn uses the value of variable x again. This is not allowed in CIF, as it creates a cyclic dependency. However, since no restrictions are introduced on the initial values of variables x, y, and z, except that they must be equal to each other, we can declare them as follows:

```
disc int x in any; // Explicit 'any' breaks the cyclic dependency.
disc int y = z;
disc int z = x;
```

Here, variable x is explicitly initialized to an arbitrary value. The other variables are initialized to be equal to whatever arbitrary value is chosen as initial value for variable x.

2.4.10. Initialization predicates

Initialization predicates can be used to specify the allowed initial locations of automata, as well as to restrict the allowed initial values of variables and initial states in general.

Initial locations of automata

Initialization predicates can be used to specify the allowed initial locations of automata:

```
automaton a:
  location loc1:
    initial;

  location loc2:
    initial true;

  location loc3;

  location loc4:
```

```
    initial false;
end
```

Automaton `a` has four locations. Location `loc1` has the `initial` keyword, and is thus allowed to be the initial location. Location `loc2` also uses the `initial` keyword, but additionally specifies a predicate that indicates under which conditions the location may be the initial location. Since it is `true`, which always holds, it does not impose any additional restrictions, and can thus always be the initial location. In fact, this is identical to location `loc1`, which did not specify a predicate, in which case it default to `true` as well.

Location `loc3` does not specify anything about initialization, and thus can never be the initial location. Location `loc4` can only be the initial location if `false` holds. Since `false` never holds, location `loc4` can never be the initial location. In fact, this is identical to location `loc3`, which did not specify any initialization at all, in which case it default to `false` as well.

Locations `loc1` and `loc2` are the potential initial locations, while locations `loc3` and `loc4` can not be chosen as initial locations of automaton `a`. Since an automaton can only have one current location, an initial location has to be chosen from the potential initial locations. In other words, the initial location of automaton `a` is either location `loc1` or location `loc2`.

Consistency between initial locations and initial values

Consider the following CIF specification:

```
automaton odd_even:
  event inc, dec;
  disc int n = 5;

  location odd:
    initial;
    edge inc do n := n + 1 goto even;
    edge dec do n := n - 1 goto even;

  location even:
    edge inc do n := n + 1 goto odd;
    edge dec do n := n - 1 goto odd;
end
```

Automaton `odd_even` keeps track of a value (`n`) that can constantly be incremented (event `inc`) and decremented (event `dec`) by one. It has two locations, that keep track of the `odd`/`even` status of value `n`. Currently, the initial value is `5`, which is odd. Therefore, the `initial` keyword is specified in the `odd` location. However, if we change the initial value of variable `n` to `6`, we have to change the initial location as well, to ensure consistent initialization. To automatically keep the initial location consistent with the initial value of variable `n`, we can change the specification to the following:

```
automaton odd_even:
```

```
  event inc, dec;
  disc int n = 5;

  location odd:
    initial n mod 2 = 1; // Initial location if 'n' is odd.
    edge inc do n := n + 1 goto even;
    edge dec do n := n - 1 goto even;

  location even:
    initial n mod 2 = 0; // Initial location if 'n' is even.
    edge inc do n := n + 1 goto odd;
    edge dec do n := n - 1 goto odd;
end
```

In this specification, location `odd` can only be the initial location if the value is odd (the value modulo two is equal to one), and location `even` can only be the initial location if the value is even. Changing the initial value of variable `n` then also changes the potential initial locations. Since the value is always odd or even, and can't be both odd and even, automaton `odd_even` always has exactly one potential initial location.

Restricting initialization

Initialization predicates can also be used to restrict the initial values of variables. And more generally, they restrict the possible initial states. It is for instance also possible to specify which combinations of locations of automata and values of variables are allowed as initial states.

As an example of restricting the allowed initial values of variables, consider the following CIF specification:

```
automaton a:
  disc int x in any;

  initial x mod 2 = 1;

  location ...
end
```

In this partial automaton, variable `x` can be initialized to any integer value, as indicated by its `int` type and the `any` keyword. However, the initialization predicate states that initially, the value of `x` module two must be equal to one. That is, the value of variable `x` must initially be odd.

It is allowed to specify initialization predicates inside automata, but it is also allowed to place them outside the automata:

```
automaton a:
  disc int x in any;
```

```
  location ...
end

automaton b:
  disc int x in any;

  location ...
end

initial a.x = 2 * b.x;
```

Here, two automata each declare a variable that can have arbitrary initial values. The initialization predicate specifies that the initial value of variable `x` from automaton `b` must be twice the initial value of variable `x` from automaton `a`.

Similarly, the initial locations of two automata can be restricted using additional initialization predicates:

```
automaton a:
  location a1:
    initial;
    ...

  location a2:
    initial;
    ...
end

automaton b:
  location b1:
    initial;
    ...

  location b2:
    initial;
    ...
end

initial (a.a1 and b.b1) or (a.a2 and b.b2);
```

The two automata, `a` and `b`, each have two potential initial locations. The initialization predicate outside the automata only allows certain combinations of initial locations of the two automata. It specifies that if automaton `a` starts in location `a1`, then automaton `b` must start in `b1`. Alternatively, if automaton `a` starts in location `a2`, then `b` must start in `b2`.

Note that locations of automata that are not indicated as being potentially initial are never initial locations. The initialization predicates outside the automata further restrict initialization, but can never make locations initial that were not already indicated as such.

This last example can however be more simply specified using only initialization predicates in locations:

```
automaton a:
  location a1:
    initial;
    ...

  location a2:
    initial;
    ...
end

automaton b:
  location b1:
    initial a.a1;
    ...

  location b2:
    initial a.a2;
    ...
end
```

The initialization for automaton `a1` remains as it was before. Instead of restricting the initialization combinations of the two automata using an extra initialization predicate, these restrictions are not put in the initialization predicates of the locations of automaton `b`. Location `b1` can only be the initial location of automaton `b` if automaton `a` is in location `a1`. And similarly, location `b2` can only be its initial location if automaton `a` starts in `a2`.

It is generally recommended to use initialization predicates in locations, where possible. If initialization must be further restricted by an initialization predicates outside locations, it is recommended to place it inside an automaton if the condition only applies to declarations from that automaton, and to place it outside of the automata if the condition applies to declarations of multiple automata.

2.4.11. Using locations as variables

Consider the following CIF specification:

```
automaton machine1:
  event start1, done1, reset1;
  disc bool claimed = false;

  location idle:
    initial;
    edge start1 when not machine2.claimed do claimed := true goto processing;

  location processing:
    edge done1 do claimed := false goto cool_down;
```

```
  location cool_down:
    edge reset1 goto idle;
end

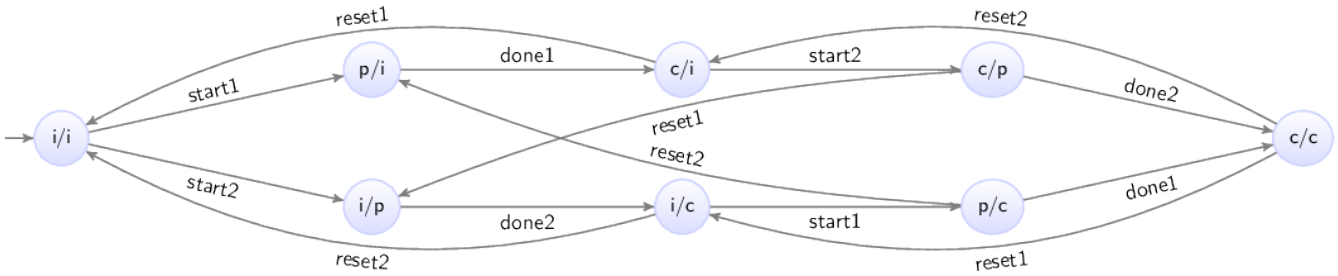
automaton machine2:
  event start2, done2, reset2;
  disc bool claimed = false;

  location idle:
    initial;
    edge start1 when not machine1.claimed do claimed := true goto processing;

  location processing:
    edge done1 do claimed := false goto cool_down;

  location cool_down:
    edge reset1 goto idle;
end
```

This specification models two machines, which produce products. The machines share a common resource, which may only be used by at most one of them, at any time (see mutual exclusion). Initially, the machines are `idle`. Then, they warm themselves up. Once they start processing, they set their boolean variable `claimed` to `true` to indicate that they claimed the shared resource. After processing, the machines release the resource, by setting `claimed` to `false`. They finish their processing cycle by cooling down, before starting the cycle for the next product. To ensure that a machine can not claim the resource if the other machine has already claimed it, the edges going to the `processing` locations have a guard that states that it is only allowed to claim the resource and start processing, if the other machine has not already claimed the resource. The state space of this specification is:



The states are labeled with the first letters of the names of the current locations of the automata.

The specification can alternatively be modeled as follows:

```
automaton machine1:
  event start1, done1, reset1;

  location idle:
    initial;
    edge start1 when not machine2.processing goto processing;
```

```
  location processing:
    edge done1 cool_down;

  location cool_down:
    edge reset1 goto idle;
end

automaton machine2:
  event start2, done2, reset2;

  location idle:
    initial;
    edge start1 when not machine1.processing goto processing;

  location processing:
    edge done1 cool_down;

  location cool_down:
    edge reset1 goto idle;
end
```

The `claimed` variables and corresponding updates have been removed, and the guards no longer use those variables. Instead, the edge for the `start1` event now refers to the `processing` location of automaton `machine2`. The guard states that the first machine can perform the `start1` event, only if the second machine is not currently in its `processing` location. In other words, the guard states that the first machine can start processing as long as the second machine is not currently busy processing (and thus using the shared resource).

The `processing` location of automaton `machine2` is used as a boolean variable. Using the location as a variable saves having to declare another variable (`claimed`) that essentially holds the same information, and needs to be explicitly updated (on two separate edges) to the correct value.

2.4.12. State (exclusion) invariants

The lesson on discrete variables used the following CIF specification:

```
automaton counter:
  event increment, decrement;

  disc int count = 3;

  location:
    initial;
    edge decrement when count > 0 do count := count - 1;
    edge increment when count < 5 do count := count + 1;
end
```

The counter can repeatedly be incremented and decremented by one, as long as the count remains at least one and at most five. To keep the count in the allowed range of values, guards were used to

limit the occurrence of the `increment` and `decrement` events.

Instead of using guards, it is also possible to use state (exclusion) invariants, also called state invariants, or just invariants:

```
automaton counter:
  event increment, decrement;

  disc int count = 3;

  invariant count >= 0;                    // Added invariants
  invariant count <= 5;

  location:
    initial;
    edge decrement do count := count - 1; // No more guards
    edge increment do count := count + 1;
end
```

The guards on the edges have been replaced by the two invariants. The first invariant specifies that the value of variable `count` must always be at least zero. The second invariant specifies that the value must also be at most five.

Invariants specify conditions that must always hold. Invariants must hold in the initial state, and all states reached via transitions. If a transition results in a state where an invariant doesn't hold, the transition is not allowed and can't be taken.

For the counter example, initially the `count` is `3`. The edge for the `increment` event can be taken, leading to a state where the `count` is `4`. Taking another transition for the `increment` event leads to a state where the `count` is `5`. If we then were to take another transition for the `increment` event, the `count` would then become `6`. However, that violates the invariant. Therefore, in the state where the `count` is `5`, no transition for the `increment` event is possible. In other words, the invariant disables the transition for the `increment` event for that state.

The two invariants can be specified in various ways:

```
// Multiple invariants, each with a single predicate.
invariant count >= 0;
invariant count <= 5;

// Single invariant, with multiple predicates.
invariant count >= 0, count <= 5;

// Single invariant, with single predicate.
invariant count >= 0 and count <= 5;
```

Each of these variants leads to the exact same behavior, and which variant to use depends mostly on the modeler's own preference.

The benefit of guards over invariants is that they more explicitly state the condition under which an edge can lead to a transition. If a guard doesn't hold, the edge can't be part of a transition. It is thus immediately clear when the edge can lead to a transition. For invariants, the update has to be calculated first, after which the invariants can be evaluated for the state resulting from the transition. If one of the invariants doesn't hold, the transition is not allowed. In the case of the invariants, it is not as immediately clear from the edge alone, when that specific edge can or can not lead to a transition.

The benefit of invariants over guards is that they apply to all edges. If several edges in an automaton have updates to the same variable, then the invariants need to be specified only once, and apply to all transitions, for all edges. Using guards, all the edges that modify the variable would need their own guards, and if the updates are different, the different edges usually require different guards. Furthermore, if new edges with updates to the same variables are added, the invariant is already present, but guards have to be added, which can easily be forgotten. In those cases, invariants can thus help keep the specification consistent.

Another benefit of invariants is that they explicitly state the conditions that must hold in relation to the variables, while guards specify the condition under which the update is allowed. Consider the following CIF specification:

```
// Using invariants.
automaton a:
  disc int x;

  invariant 0 <= x, x <= 100;

  location:
    initial;
    edge do x := 2 * x + 3;
end

// Using guards.
automaton a:
  disc int x;

  location:
    initial;
    edge when x <= 48 do x := 2 * x + 3;
end
```

The goal is to keep the value of in the range [0..100]. The invariant is simple and direct. The guard however, has to state the condition under which the update does not violate the goal. That is, the upper bound has to be decreased by three, and the result has to be divided by two, to get the highest value (48) for which the update is still within the valid range of values. That is, for value 48 the update results in value 99 (2 * 48 + 3 = 99), and for value 49 the update results in value 101 (2 * 49 + 3 = 101). The more complex the update, the harder it is to figure out the guard to use to keep satisfy the goal.

You can of course also use `2 * x + 3 <= 100` as guard, instead of `x <= 48`. However, this duplicates

part of the update in the guard.

So far, all invariants have been specified in automata. They may however also be specified outside of the automata, similar to initialization predicates. It is generally recommended to place an invariant inside an automaton if the condition only applies to declarations from that automaton, and to place it outside of the automata if the condition applies to declarations of multiple automata.

Furthermore, invariants can be placed in a location. Such an invariant only has to hold while the location in which it is specified is the current location of its automaton.

Naming state (exclusion) invariants

In some cases, it might be useful to give names to invariants. It may improve the readability of the model and it makes it easier to refer to them. Names can be given as follows:

```
// Multiple named invariants.
invariant nonNegativeCount: count >= 0;
invariant maximumCount: count <= 5;

// Shorter notation for multiple named invariants.
invariant nonNegativeCount: count >= 0, maximumCount: count <= 5;
```

Invariant names must be unique.

2.4.13. State/event exclusion invariants

Consider an elevator, consisting of three parts: a motor to make the elevator move up and down, a door that can be opened and closed to let passengers enter and exit, and an emergency button that can be used to stop the elevator in case of an emergency. The following CIF specification models the three parts:

```
automaton motor:
  event turn_on, turn_off;

  location off:
    initial;
    edge turn_on goto on;

  location on:
    edge turn_off goto off;
end

automaton door:
  event open, close;

  location closed:
    initial;
    edge open goto opened;
```

```
  location opened:
    edge close goto closed;
end

automaton emergency_button:
  event push, release;

  location released:
    initial;
    edge push goto pushed;

  location pushed:
    edge release goto released;
end
```

Each part is modeled by an automaton. Since the automata don't share any events, they operate independently. What is missing, is a controller that links the different automata, and controls them in a safe manner. Such a controller restricts the behavior of the individual automata, allowing only the combined behavior that is deemed desired. There are several ways to restrict events, including introducing synchronization between the different automata, and adding guards. The downside of these approaches it that they require modification of the automata. What if we wanted to specify the controller separately from the behavior of the physical system? We could introduce an additional automaton, that synchronizes with the existing automata. For instance, we could add the following to the CIF specification:

```
automaton controller:
  location:
    initial;
    edge motor.turn_on when door.closed and emergency_button.released;
end
```

This controller introduces restrictions for the `turn_on` event of the `motor`. In this particular case, the controller ensures that the motor may only be turned on when both the door is closed and the emergency button is released. By restricting the event, the controller prohibits the event from taking place in certain states, ensuring that only the desired behavior remains.

It is nice that we can separate the description of the physical behavior of the elevator from the controller that controls it. This separation of concerns may make it easier to reason about the behavior, it may make it easier to adapt the controller when the physical system doesn't change, and it may make it easier to reuse the model of the physical system for other purposes.

However, modeling an automaton with a single location that must then also be `initial` requires quite some syntax. State/event exclusion invariants can serve the same purpose, but are often easier to use, shorter to write, and more intuitive to read. Instead of the `controller` automaton, we can also use the following:

```
invariant motor.turn_on needs door.closed and emergency_button.released;
```

Each state/event exclusion invariant restricts an event, preventing it from happening in certain states. That is, the event is excluded from taking place in certain states. In this case, the `turn_on` event of the `motor` automaton needs the `door` to be in its `closed` location and the `emergency_button` to be in its `released` location, for the event to be allowed/enabled. For the states in which that condition doesn't hold, the event is disabled.

The invariant consists of two conditions. It can also be written as two separate state/event exclusion invariants, one for each condition:

```
// Single state/event exclusion invariant.
invariant motor.turn_on needs door.closed and emergency_button.released;

// Multiple state/event exclusion invariants.
invariant motor.turn_on needs door.closed;
invariant motor.turn_on needs emergency_button.released;
```

The second and third invariants lead to the same behavior as the first combined invariant. The second invariant ensures that the event can only take place when the door is closed, while the third invariant ensures that the event can only take place when the emergency button is released. The second and third invariants each indicate a necessary condition that must hold for the event to be allowed/enabled. Together, they require that both conditions hold, for the event to be allowed/enabled. If one of the conditions doesn't hold, the event will be disabled.

The `door` is either `opened` or `closed`. So far, we've required that the door is closed to allow the motor to be turned on. We can also specify it the other way around: to disallow the motor to be turned on, while the door is opened:

```
// State/event exclusion invariant to specify when event is allowed/enabled.
invariant motor.turn_on needs door.closed;

// State/event exclusion invariant to specify when event is disallowed/disabled.
invariant door.opened disables motor.turn_on;
```

Both invariants have the exact same effect. The first invariant only allows the motor to be turned on while the door is closed, which means that it disallows the motor to be turned on in all other situations, namely when the door is opened. And that is exactly what is specified by the second invariant: when the door is opened, turning the motor on is disallowed/disabled. In general, state/event exclusion invariants can always be specified as a positive form (allowed/enabled) and a negative form (disallowed/disabled). It is up to the modeler to choose, based on considerations such as personal preference and readability. Consider the following four alternative forms:

```
// Single state/event exclusion invariant for enabling the event.
invariant motor.turn_on needs door.closed and emergency_button.released;
```

```
// Multiple state/event exclusion invariants for enabling the event.
invariant motor.turn_on needs door.closed;
```

```
invariant motor.turn_on needs emergency_button.released;
```

```
// Single state/event exclusion invariant for disabling the event.
invariant door.opened or emergency_button.pushed disables motor.turn_on;
```

```
// Multiple state/event exclusion invariants for disabling the event.
invariant door.opened             disables motor.turn_on;
invariant emergency_button.pushed disables motor.turn_on;
```

Each of the four forms has the exact same effect, but is written in a different way.

We already saw earlier that for state/event exclusion invariants that introduce necessary conditions for an event to be enabled (the `needs` variant), the conditions can be combined using an `and` operator to form a combined condition, for a single invariant.

Here, we also see how in a similar way, state/event exclusion invariants that introduce sufficient conditions for an event to be disabled (the `disabled` variant) can be combined. Each of them individually has a condition, that if satisfied disables the event, regardless of the other invariants. So, if one of them disables the event, the event is disabled. To combine such invariants into a single invariant, the conditions need to be combined using an `or` operator, as shown above.

It may occur that multiple events need to be disabled for the same conditions. Instead of writing multiple invariants with the same conditions, one for each event, they can also be combined:

```
// Two separate invariants with same condition, for different events.
invariant motor.turn_on needs emergency_button.released;
invariant door.close    needs emergency_button.released;

// Combined invariant, for multiple events.
invariant {motor.turn_on, door.close} needs emergency_button.released;
```

The first two invariants have the same condition, but restrict different events. The third invariant has the same condition, but restricts both events. In general, for all state/event invariants, multiple events may be given, if they share the same condition. The events must then be separated by spaces and be enclosed in curly brackets (`{...}`).

Naming state/event exclusion invariants

In some cases, it might be useful to give names to invariants. It may improve the readability of the model and it makes it easier to refer to them. Names can be given as follows:

```
invariant controller: motor.turn_on needs door.closed and emergency_button.released;
```

Invariant names must be unique. It is not supported to give a name to a combined invariant that applies to multiple events.

2.5. Types and values

2.5.1. Types, values, and expressions

This lesson explains the difference between types, values, and expressions. These concepts have already been used in previous lessons, but this lesson names them explicitly, and also explains the relations between them. Furthermore, this lesson serves as an introduction for the coming lessons, which rely heavily on these concepts. Consider the following declarations of discrete variables:

```
disc int x = 1;
disc int y = 2 * x;
```

The first declaration declares a discrete variable named `x`, and the second declaration declares a discrete variable named `y`. Both variables have an `int` *data type*. A data type is usually just called a *type*, if there is no confusion with other kinds of types. The type of a variable indicates the potential or allowed values of the variable. Variable `x` is initialized to *value* `1`. Variable `y` is initialized to twice the value of `x`, meaning it is initialized to value `2`.

Both `1` and `2 * x` are *expressions*. Expressions are combinations of among others literal values (e.g. `1`), variables (e.g. `x`), and operations (e.g. `*`) on them. Expressions can be *computed*, resulting in a value. This is called *evaluation* of the expression.

Expression `2 * x` can be evaluated. Evaluating the expression results in value `2` if the value of `x` is `1`, and in value `4` if the value of `x` is `2`. Expressions can thus be evaluated to different values, depending on the values of the variables that occur in them.

Expression `1` consists of only a single value, called a *literal* value expression. Evaluation always results in that single value. Expression `1 + 3` evaluates to value `4`. Even though it does not consist of only just a literal, the value is the same for each evaluation. The expression represents a *constant* value.

2.5.2. Values overview

The values of CIF (and their types) can be categorized into different categories: elementary values, container values, and miscellaneous values. Elementary values represent single values, such as a single number. Container values represent multiple values. The different container values combine or store the values in different ways. The elementary and container values are described in the remainder of this part of the tutorial. The miscellaneous values are special, and are explained later in the tutorial. The remainder of this lesson gives an overview of the values available per category, along with short descriptions of each of the different kinds of values.

Elementary values

Booleans

Represents truth values of for instance guards and other conditions. The only two possible values are `true` and `false`.

Integers

Represent integer numbers, such as `-123` and `5`.

Enumerations

Represent enumerated values, which are collections of names each representing a different value. For instance, a enumeration named `color` could have values `red`, `green`, and `blue`.

Reals

Represent real numbers, such as `1.56` and `-2.7e6` (scientific notation for 2.7 million).

Strings

Represent textual values, such as `"hello world"` and `"some text"`.

Container values

Tuples

Tuples have two or more ordered values, each of which can have a different type. For instance: `(1, true, 5.0)`.

Lists

Lists have zero or more ordered values, each of which has the same type, and possibly with duplicates. For instance: `[1, 5, 2, 1]`.

Sets

Sets have zero or more unordered values, each of which has the same type, and without any duplicates. For instance: `{1, 5, 2}`.

Dictionaries

Dictionaries have keys and associated values. The keys are unique and each map to a value. For instance: `{1: true, 2: false, 3: false}`.

Miscellaneous values

Functions

Functions take values and use them to compute other values, possibly using complex and lengthy calculations.

Distributions

Stochastic distributions allow for sampling, making it possible to produce random values.

2.5.3. Integers

Integers are whole numbers, numbers without a fractional part. Examples include `-123` and `5`. The default value of integers (`int` type) is `0`. Several standard arithmetic operators and functions are available to work with integers, including the following:

```
+9          // 9
--9         // 9
```

```
9 + 4       // 13
9 - 4       // 5
9 * 4       // 36
9 / 4       // 2.25 (result is a real number, not an integer number)

9 div 4     // 2    (9 / 4 = 2.25, so 4 fits at most two whole times in 9)
9 mod 4     // 1    (the remainder of 9 div 4)

pow(2, 4)   // 16   (2 to the power of 4, or 2 * 2 * 2 * 2)
abs(-9)     // 9    (absolute value)
min(9, 4)   // 4    (minimum value)
max(9, 4)   // 9    (maximum value)
```

Integer values can be compared to other integer values:

```
x < y       // less than
x <= y      // less than or equal to
x = y       // equal to
x != y      // not equal to
x >= y      // larger than or equal to
x > y       // larger than
```

The next lesson explains how to define ranged integers.

2.5.4. Ranged integers

CIF can only represent integer numbers (type `int`) in the range -2,147,483,648 (= -2^{31}) to 2,147,483,647 (= $2^{31} - 1$). Using values outside that range results in the CIF model being invalid, and leads to runtime errors:

```
ERROR: Integer overflow: 9999999 * 9999999.
```

It is possible in CIF to explicitly specify that only a sub range of the integer values are allowed:

```
disc int[3..7] x;
```

This variable `x` can only have integer values that are at least `3` and at most `7`. Assigning any other value to `x` is not allowed. The default value is not `0` but `3`, as that is the value closest to `0` that is in the allowed range of values.

2.5.5. Reals

Reals or real numbers are numbers with a fractional part. Examples include `1.56` and `-2.7e6` (scientific notation for 2.7 million). Real numbers must either have a fraction or use the scientific notation, to distinguish them from integer numbers. The default value of reals (`real` type) is `0.0`.

Several standard arithmetic operators and functions are available to work with reals, including the following:

```
+1.23           // 1.23
--1.2           // 1.2

1.5 + 0.5       // 2.0
1.5 - 0.5       // 1.0
1.5 * 0.5       // 0.75
1.5 / 0.5       // 3.0

pow(3.5, 2.0)   // 7.0      (3.5 to the power of 2, or 3.5 * 3.5)
abs(-1.5)       // 1.5      (absolute value)
min(1.5, 0.5)   // 0.5      (minimum value)
max(1.5, 0.5)   // 1.5      (maximum value)

sqrt(16.0)      // 4.0      (square root)
cbrt(16.0)      // 2.0      (cube root)

sin(1.0)        // 0.841... (sine)
cos(1.0)        // 0.540... (cosine)
tan(1.0)        // 1.557... (tangent)

log(100.0)      // 2.0      (logarithm to base 10)
ln(100.0)       // 4.605... (natural logarithm)
```

Real values can be compared to other real values, as well as to integer values:

```
x < y       // less than
x <= y      // less than or equal to
x = y       // equal to
x != y      // not equal to
x >= y      // larger than or equal to
x > y       // larger than
```

Integer numbers can often be written where real numbers are expected. Real values and integer values can also often be combined using arithmetic operators and functions. Furthermore, it is possible to convert between them, e.g as follows:

```
sqrt(16)    // 4.0 (16 interpreted as 16.0)
1 + 0.5     // 1.5 (addition of an integer number and a real number)
max(0.5, 1) // 1.0 (maximum of an integer number and a real number)

<real>1     // 1.0 (cast from integer to real, explicit conversion)
round(1.6)  // 2   (round real to closest integer, half up)
ceil(0.7)   // 1   (round real up to integer)
floor(0.7)  // 0   (round real down to integer)
```

2.5.6. Booleans

Booleans represents truth values of for instance guards and other conditions and properties. The only two possible values are `true` (condition or property holds) and `false` (condition or property does not hold). The default value of booleans (`bool` type) is `false`. Several standard logical operators are available to work with booleans, including the following:

```
not x       // inverse

x and y     // conjunction (both x and y must hold)
x or y      // disjunction (either x, y, or both must hold)
x => y      // implication (if x holds, y must hold)

x = y       // equal to
x != y      // not equal to
```

The condition `x < 3` evaluates to `true` if `x` is less than `3` and to `false` if `x` is `3` or larger than `3`. The result is thus a boolean value.

Conditions can be combined. `x >= 3 and x <= 9` means that the value of `x` must be both at least `3` and at most `9`. `x >= 3 or x <= 9` means that the value of `x` must be at least `3`, at most `9`, or both. Since the condition is always satisfied (it always evaluates to `true`), condition `true` can be used instead of `x >= 3 or x <= 9`.

2.5.7. Strings

Strings represent textual values, as a sequence of characters. String values are always written between double quotes. An example is `"hello world"`. The default value of strings (`string` type) is the empty string `""`.

Strings can be composed using the `+` operator. The expression `"hello" + " " + "world"` evaluates to `"hello world"`. For advanced text formatting, see the later lesson on text formatting.

2.5.8. Enumerations

Enumerations represent collections of related entities, such as types of products, types of available resources, available machine types, different countries, different colors, different genders, and so on. It is possible to use numbers to represent the different entities, for instance `0` for red, `1` for orange, and `2` for green, to represent the different colors of a traffic light. However, these numbers are rather arbitrary. Furthermore, they don't actually represent numbers, but rather they represent one of the entities (red, orange, green). Enumerations allow giving each entity a name, and to use those names instead of numbers. This usually makes the model easier to read and understand. For instance, consider the following:

```
enum TrafficColor = RED, ORANGE, GREEN;
```

The `enum` keyword is used to declare an enumeration. The `TrafficColor` enumeration has three possible values or *literals*. The literals are named `RED`, `ORANGE`, and `GREEN`. An enumeration can be used as data type, and the enumeration literals can be used as values:

```
disc TrafficColor light = RED;
```

The `TrafficColor` enumeration is used as type of the `light` variable. The `light` variable is given value `RED` as its initial value. The default value of an enumeration type is its first literal (`RED` in this case). However, it is usually preferred to explicitly initialize variables with enumeration types, for readability.

```
edge change_color when light = RED do light := GREEN;
```

This edge has a guard that compares the value of the `light` variable to enumeration literal `RED`. Only if the `light` is currently `RED`, may this edge be taken. The edge further assigns enumeration literal `GREEN` as the new value of variable `light`. The edge as a whole models that if the `light` is currently `RED`, it may change color (event `change_color`) and become `GREEN`.

2.5.9. Tuples

Tuples are used for keeping several (related) kinds of data together in one variable, e.g. the identification number and weight of a box. A tuple consists of a number of fields, where the types of these fields may be different. The number of fields is fixed. For instance, consider the following:

```
disc tuple(int nr; real weight) box;
```

Variable `box` has a tuple type, consisting of two fields, an integer typed field with name `nr` and real typed field with name `weight`. The `box` variable has essentially two values, an integer typed value, and a real typed value.

If multiple consecutive fields have the same type, the type need not be repeated for each of them. In the following example, variables `x1` and `x2` have the same type:

```
disc tuple(int a; int b; real c; int d) x1;
disc tuple(int a, b;     real c; int d) x2;
```

Literal values exist for tuple types:

```
disc tuple(int nr; real weight) box = (5, 2.7);

edge ... do box := (6, 3.4);
```

The `box` variable is initialized to a tuple value consisting of integer value `5` (identification number) and real value `2.7` (weight). The entire value of the variable is reassigned in the assignment. That is,

both fields are given new values.

It is also possible to refer to a specific field of a tuple, using *projection*:

```
disc tuple(int nr; real weight) box = (5, 2.7);
disc int i;
disc real r;

edge ... do i := box[nr];            // Projection to field 'nr'.
edge ... do r := box[weight];        // Projection to field 'weight'.
edge ... do box[nr] := i;            // Assignment to field 'nr'.
edge ... do box[nr] := box[nr] + 1;  // Increment of field 'nr'.
```

The first edge uses projection to obtain the value of the integer `nr` field, and assign it to integer variable `i` (`i` becomes `5`). The second edge performs a similar operation for the `weight` field (`r` becomes `2.7`). The third edge assigns the value of integer variable `i` to the integer field `nr` of the `box` variable. This changes only the value of the `nr` field. The value of the `weight` field of the `box` variable is not affected by this assignment. The third edge increments the value of the `nr` field of the `box` variable by one, leaving the `weight` of the `box` as is. Besides projection using field names, it is also possible to do projection using 0-based indices:

```
disc tuple(int nr; real weight) box = (5, 2.7);
disc int i;
disc real r;

edge ... do i := box[0];  // Projection to field 'nr'.
edge ... do r := box[1];  // Projection to field 'weight'.
```

Index `0` refers to the first field, in this case field `nr`. Index `1` refers to the second field, etc. Projection using indices is also called *indexing*. For tuples, it is usually preferred to use field names, rather than indices, for readability.

It is possible to create a tuple from separate values, each stored in a variable:

```
disc tuple(int nr; real weight) box;
disc int i;
disc real r;

edge ... do box := (i, r); // Packing a tuple.
```

The right hand side of the assignment is a tuple literal value, as used before. The field values however, are obtained by evaluation of variables, rather than using literal integer and real values. This kind of assignment, where there is tuple variable at the left hand side, and values for each of the fields of that tuple at the right hand side, is called *packing* a tuple.

It is possible to obtain the values of the fields of a tuple into separate variables:

```
disc tuple(int nr; real weight) box = (5, 2.7);
disc int i;
disc real r;

edge ... do i := box[nr], r := box[weight];
edge ... do (i, r) := box; // Unpacking a tuple.
```

The first edge uses projection on the variable `box` to obtain the values of the individual fields, and assigns those extracted values to two separate variables. The second edge does the same thing as the first edge, and is preferred in this case, because of its simple and short notation. This kind of use, where at the left hand side of the assignment you see variables for each of the fields of the tuple, and on the right hand side you see only one variable that has a tuple type, is called *unpacking* a tuple.

2.5.10. Lists

A list is an ordered collection of values (called *elements*) of a same type. Lists can be used to model anything where duplicate values may occur or where order of the values is significant. Examples are customers waiting in a shop, process steps in a recipe, or products stored in a warehouse. Consider the following:

```
disc list int x = [7, 8, 3];
```

Variable `x` has a list of integers as its value. In this case, its initial value is a literal list with three integer elements. Lists are ordered collections of elements. `[7, 8, 3]` is thus a different list as `[8, 7, 3]`. Lists are empty by default, and they may have duplicate elements:

```
disc list int x1;                       // Implicitly empty list.
disc list int x2 = [];                  // Explicitly empty list.
disc list int x3 = [1, 2, 1, 2, 2];     // Duplicate elements in a list.
```

Projection/indexing

Since lists are ordered, there is a first element and a last element (unless the list is empty). An element can be obtained by *projection*, usually called *indexing* for lists:

```
disc list int x = [7, 8, 3];
disc int i;

edge ... do i := x[0];   // 'i' becomes '7'
edge ... do i := x[1];   // 'i' becomes '8'
edge ... do i := x[2];   // 'i' becomes '3'
```

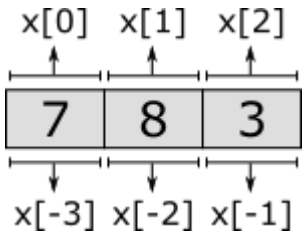
```
edge ... do i := x[3];   // error (there is no fourth element in the list)
edge ... do x[0] := 5;   // the first element of 'x' is replaced by '5'
```

The first three edges obtain specific elements of the list, and assign them to variable `i`. The first element is obtained using *index* or *offset* `0`, the second element using index `1`, etc. The index of the last element is the length of the list (the number of elements in the list), minus one. Indexing does not change the value of variable `x`. The fourth edge is invalid, as the fourth element (index `3`) of variable `x` is used, which does not exist. The fifth edge replaces only the first element (index `0`) of list `x`, while keeping the remaining elements as they are. It is also allowed to use negative indices:

```
disc list int x = [7, 8, 3];
disc int i;

edge ... do i := x[-1];  // 'i' becomes '3'
edge ... do i := x[-2];  // 'i' becomes '8'
edge ... do i := x[-3];  // 'i' becomes '7'
edge ... do i := x[-4];  // error (there is no element before element '7')
```

Negative indices start from the back of the list, rather than from the front. Index `-1` thus always refers to the last element, `-2` to the second to last element, etc. As with the non-negative indices, using a negative index that is out of range of available elements, results in an error. To obtain a non-negative index from a negative index, add the negative index to the length of the list: `3 + -1 = 2`, `3 + -2 = 1`, and `3 + -3 = 0`. The following figure visualizes a list, with non-negative indexing (at the top) and negative indexing (at the bottom):



Slicing

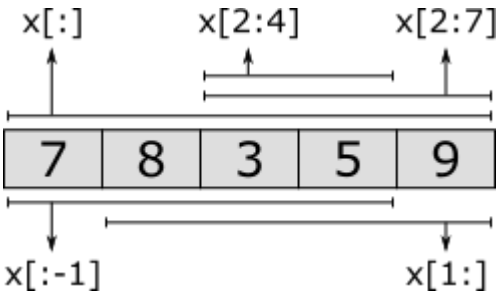
Besides obtaining a single element from a list, it is also possible to obtain a sub-range of the elements of a list, called a *slice*. Slicing also does not change the contents of the list. It results in a copy of a contiguous sub-sequence of the list. The result of a slice operation is again a list, even if the slice contains just one element, or no elements at all. Slicing requires two indices: the index of the first element of the sub-range (inclusive), and the index of the last element of the sub-range (exclusive). Both indices may be omitted. If the start index is omitted, it defaults to zero. If the end index is omitted it defaults to the length of the list. If the begin index is equal to or larger than the end index, the slice is empty. Similar to indexing, negative indices may be used, which are relative to the end of the list rather than the start of the list. Indices that are out of bounds saturate to those bounds. Some examples:

```
disc list int x = [7, 8, 3, 5, 9];

x[2:4]  //       [3, 5]    Slice that includes third and fourth elements.
```

```
x[2:7]  //       [3, 5, 9] Slice that excludes the first two elements.
x[1:]   //    [8, 3, 5, 9] Slice that excludes the first element.
x[:-1]  // [7, 8, 3, 5]    Slice that excludes the last element.
x[:]    // [7, 8, 3, 5, 9] Slice includes all elements (is equal to 'x').
```

The first slice takes the third (index `2`) and fourth (index `3`) elements. The begin index (`2`) is thus included, the end index (`4`) is not. The second slice starts at the same index, but extends to the sixth element (index `7`). Since there are only five elements, the index is saturated (or clamped) to the end of the list. The results is that all but the first two elements are included. The third slice excludes the first element (index `0`), by starting at index `1`. It omits the end index, meaning that the entire remainder of the list is kept, and only the first element is not included. The fourth slice begins at the beginning of the list, as the begin index is omitted. It continues until the last element of list, which it excludes. It thus excludes a single element from the end of the list. The fifth slice includes all elements, as both the begin and end index are omitted. The slice is thus identical to the list in `x`. The following figure graphically represents the slices:



Concatenation

Lists can be combined into new lists. They are essentially 'glued' together. This is called *concatenation*. This can also be used to add a single element to the front or back of the list. For instance:

```
[7, 8, 3] + [5, 9]   // [7, 8, 3, 5, 9]
[5] + [7, 8, 3]      // [5, 7, 8, 3]
[7, 8, 3] + [5]      // [7, 8, 3, 5]
```

List comprehensions

List comprehensions can be used to perform advanced manipulations of lists. For instance, consider the following example:

```
[x + 1 for x in lst if x >= 0]
```

This list comprehension takes a list `lst`, considers each non-negative element `x`, increments `x` by one, and includes it in the result.

In general, a list comprehension contains a *source* expression, listed after the `in` keyword. It is evaluated to get the list to manipulate. An *addressable*, specified after the `for` keyword, is used to name the elements being manipulated. If an optional *filter* expression is specified, using the `if`

keyword, only elements that satisfy the filter expression are considered. Finally, a *result* expression is specified at the start of the list comprehension. The result expression is evaluated for each element that is not filtered out, producing an element of the resulting list.

While an addressable can be a single identifier, as in the example above, it can also be used to unpack tuples. For instance, consider the following example:

```
[a + b for a, b in some_tuple_list]
```

In this case, `some_tuple_list` is a list of tuples, where the first field is unpacked into `a` and the second field into `b`. Here, the two local variables are added together to produce an element of the resulting list.

Other operations

Several other standard operators and functions are available to work with lists, including the following:

```
[1, 8, 3] = [1, 3, 8]          // false (equality test)
6 in [1, 8, 3]                 // false (element test)
1 in [1, 8, 3]                 // true

empty([1, 2])                  // false (empty test)
size([1, 5, 3, 3])             // 4 (count elements)

del([7, 8, 9, 10], 2)          // [7, 8, 10] (removed value at index '2')
pop([1, 5, 3])                 // (1, [5, 3]) (first element and remainder)
```

2.5.11. Bounded lists and arrays

CIF can only represent lists (type `list`) with at most 2,147,483,647 (= 2^{31} - 1) elements. Using lists with more elements results in the CIF model being invalid, and leads to runtime errors. For instance, consider the following CIF specification:

```
automaton a:
  disc list int x;

  location:
    initial;
    edge do x := x + [1];
end
```

Each time the edge is taken, another element is added to list `x`. As soon as an attempt is made to add the 2,147,483,648th element, a runtime error occurs.

It is possible to explicitly restrict the number of elements that may be contained in a list:

```
disc list[3..7] int y;      // List with at least 3 and at most 7 elements.
```

Variable `y` can only have lists as its value that have at least `3` and at most `7` elements. Assigning a list with any other number of elements is not allowed. Lists with size restrictions are called *bounded lists*. They can also be called *size restricted lists* or *ranged lists*. The default value for `y` is `[0, 0, 0]`. That is, the default value has the least amount of elements that is allowed by the bounded list, and the default value (`0`) of the element type (`int`).

Lists with a fixed length are called *arrays*:

```
disc list[5..5] int a;      // List with at least 5 and at most 5 elements.
disc list[5]    int b;      // Shorter but equivalent array notation.
```

By giving a bounded list the same lower and upper bound, the bounded list has a fixed number of elements, and can be called an *array*. Arrays also have a shorter and more convenient notation, where the number of elements is only given once. Both notations for arrays are equivalent.

Both bounded lists and arrays support the same operations as regular lists, and can be modified (assigned) the same way as regular lists, as long as their size restrictions are not violated.

2.5.12. Sets

A set is an unordered collection of values (called *elements*) of a same type. Each element value either exists in a set, or it does not exist in a set. Each element value is unique, as duplicate elements are silently discarded. Consider the following:

```
disc set int x1 = {3, 7, 8};
disc set int x2 = {8, 3, 7};        // Order irrelevant (same as 'x1').
disc set int x3 = {8, 3, 7, 3};     // Duplicates ignored (same as 'x2').
disc set int x4 = {};               // Empty set.
```

Variable `x1` has a set of integers as its value. In this case, its initial value is a literal set with three integer elements. As sets are unordered collections of elements, `{3, 7, 8}` is the same set as `{8, 3, 7}`, and thus variables `x1` and `x2` have the same initial values. Since elements in a set are unique, set `{8, 3, 7}` is equal to the set `{8, 3, 7, 3}`, and thus variables `x2` and `x3` have the same initial values. For readability, elements in a set are normally written in increasing order, for example `{3, 7, 8}`. Variable `x4` has an empty set as initial value, which is also the default initial value for sets.

The *union* of two sets results in a set that contains the combined elements of both sets. You can think of the resulting set containing the elements that are in the one set `or` in the other set (or in both of them). Since sets never contain duplicates, common elements are present only once in the resulting set:

```
{1, 2, 3} or {2, 3, 4}          // {1, 2, 3, 4}
```

```
{1, 2, 3} or {2, 3, 4}          // {1, 2, 4, 3}
```

Since sets are unordered, both answers are possible, and represent the same set. Since the order is irrelevant, there are 24 different ways to represent that same set. In the remainder of this lesson, we'll use increasing order, for readability.

The *intersection* of two sets results in a set that contains the elements that are present in both sets. You can think of the resulting set containing the elements that are in the one set `and` in the other set. In other words, the result contains the elements common to both sets:

```
{1, 2, 3} and {2, 3, 4}         // {2, 3}
{1, 2} and {3, 4}               // {} (no elements in common)
```

The *difference* of two sets results in a set that contains the elements of the first set that are not present in the second set. You can think of the resulting set containing the elements of the first set, with the elements of the second set subtracted or removed from it:

```
{1, 2, 3} - {2, 3, 4}           // {1}
{1, 2, 3} - {4, 5}              // {1, 2, 3}
{1, 2, 3} - {1, 2, 3, 4}        // {}
```

Several other standard operators and functions are available to work with sets, including the following:

```
{1, 8, 3} = {1, 3, 8}       // true (equality, ignores order of elements)

6 in {1, 8, 3}              // false (element test)
1 in {1, 8, 3}              // true

{1, 3} sub {1, 3}           // true (subset check)
{1, 3} sub {1, 3, 5}        // true
{1, 3} sub {1, 4}           // false
{1, 3} sub {1, 4, 5}        // false

empty({1, 2})               // false (empty test)
size({1, 5, 3, 3})          // 3 (count elements, duplicates ignored)
```

2.5.13. Dictionaries

A dictionary is an unordered collection of *keys* and associated *values*. A key with its associated value is called a key/value *pair*. Consider the following:

```
disc dict(string:int) age = {"eve": 32, "john": 34, "adam": 25};
```

Variable `age` has as value a dictionary consisting of pairs of strings (the keys) and integers (the

values). In this example, each string represents a person's name, and each integer represents the age of that person. Variable `age` is initialized with a literal dictionary, containing three key/value pairs. You can think of the dictionary as storing the information that `eve` is `32` years old, or that the age of `eve` is `32`.

As with sets, dictionaries are unordered. The order of the key/value pairs is irrelevant, `{"eve": 32, "adam": 25}` is the same dictionary as `{"adam": 25, "eve": 32}`. For readability, key/value pairs of dictionary literals are normally written in increasing order of their keys. `{"adam": 25, "eve": 32}` is thus preferred over `{"eve": 32, "adam": 25}`, as `adam` goes before `eve` in a phone book.

Dictionary literals are often written using multiple lines, to get two 'columns' for the keys and values, which can improve readability:

```
disc dict(string:int) age = {"eve":  32,
                             "john": 34,
                             "adam": 25};
```

The default value for dictionary types, is an empty dictionary. The following two variables thus have the same initial value:

```
disc dict(string:int) x1 = {};  // Explicitly initialized as empty.
disc dict(string:int) x2;       // Implicitly initialized as empty.
```

Every key of a dictionary is unique, but they may be associated with the same value. For the above example with ages, the names (keys) are used to uniquely identify people, but multiple people may have the same age (values). It is not allowed to have the same key twice, in a dictionary literal, regardless of whether they map to the same value or to different values:

```
disc dict(int:int) x1 = {1: 2, 1: 2};  // Invalid due to duplicate key '1'.
disc dict(int:int) x2 = {1: 2, 1: 3};  // Invalid due to duplicate key '1'.
```

The values of a dictionary can be obtained by *projection* on that dictionary, using the keys:

```
disc dict(string:int) age = {"adam": 25, "eve": 32, "john": 34};
disc int i;

edge ... do i := age["adam"];   // 'i' becomes '25'
edge ... do i := age["eve"];    // 'i' becomes '32'
edge ... do i := age["carl"];   // error (there is no "carl" key)
```

Projection using a key that exists in the dictionary, results in the associated value. Projection using a key that does not exist in the dictionary, leads to a runtime error.

It is possible to modify single elements of a dictionary, as follows:

```
disc dict(string:int) age = {"adam": 25, "eve": 32, "john": 34};

edge ... do age["eve"] := 33;  // Changes eve's age.
edge ... do age["bob"] := 47;  // Adds a new key/value pair.
```

The age of `eve` is changed from `32` to `33`. The age is thus replaced by a new age. The age of `bob` is set to `47`. Since there was no key/value pair for that person in the dictionary, a new key/value pair is added. After the updates of both edges, the value of variable `age` is `{"adam": 25, "bob": 47, "eve": 33, "john": 34}`.

Several other standard operators and functions are available to work with dictionaries, including the following:

```
{"a": 1, "b": 2} = {"b": 2, "a": 1}     // true (equality check)
{"a": 1, "b": 2} = {"a": 1, "b": 3}     // false

"a" in {"a": 1, "b": 2}                 // true (key existence check)
"c" in {"a": 1, "b": 2}                 // false

{"a": 1, "b": 2} + {"b": 3, "c": 4}     // {"a": 1, "b": 3, "c": 4} (add/overwrite
pairs)

{"a": 1, "b": 2} - {"b": 3, "c": 4}     // {"a": 1} (removal based on keys)
{"a": 1, "b": 2} - {"b", "c"}           // {"a": 1}
{"a": 1, "b": 2} - ["b", "c"]           // {"a": 1}

empty({"a": 1, "b": 2})                 // false (empty check)
size({"a": 1, "b": 2})                  // 2 (count pairs)
```

2.5.14. Combining values

Values of different types can usually be arbitrarily combined. For instance, consider the following example:

```
disc dict(int;tuple(real pos, weight)) boxes = {1: (0.0, 2.5),
                                                2: (3.0, 1.7),
                                                3: (4.0, 3.9};
```

Variable `boxes` stores data about multiple boxes, in a dictionary with unique identification numbers (`1`, `2`, and `3`) used as keys. For each box, the position (`pos`) and `weight` are stored as a tuple. Initially, there are three boxes. The first box has identification number `1`, position `0.0`, and weight `2.5`. The second box has identification number `2`, position `3.0`, and weight `1.7`. Also consider the following example, where the data of that variable is manipulated:

```
edge ... do boxes[1][weight] := 3.5;
edge ... do boxes[2][pos] := boxes[2][pos] + 1;
```

```
edge ... do boxes[4] := (1.0, 0.8);
```

The first edge changes the `weight` of the box `1` (the box with identification number `1`), from `2.5` to `3.5`. The second edge increases the position (`pos`) of box `2` from `3.0` to `4.0`. The third edge adds data for a new box with identification number `4`.

2.5.15. If and switch expressions

In some models one wants to produce different values under different circumstances. Or one needs to convert from one type to another type. `if` and `switch` expressions provide means to do this in a concise manner. This lesson provides an introduction to both these expressions. For more practical examples, see the simulation model of the synthesis-based engineering example elsewhere in the documentation.

If expression

Consider the following expression:

```
if x > 0: 1 elif x < 0: -1 else 0 end
```

This expression takes a real number `x` and produces its sign. The `if` expression is used to detect the different situations, and to produce the correct result in each of those situations.

An `if` expression always consist of the `if` and the `else` parts. In between can optionally be any number of `elif` parts:

```
if x > 0: 1 elif x < 0: -1 else 0 end
if x > 0: 1 else 0 end
if x > 0: 1 elif x < -10: -2 elif x < 0: -1 else 0 end
```

If the 'guard predicate' of an `if` or `elif` (the expression before the `:`) holds, that alternative is chosen. In the following example, if `x` has a value of `2` and `y` a value of `3`, then value `-2.8` is produced:

```
if x > 0 and y = 2: 1.5 elif x > 0 and y != 2 or x < 0: -2.8 else 0.1 end
```

The `if` and `elif` alternatives are considered in order. The first alternative that has a guard that holds determines what value is produced by the `if` expression. Consider the following `if` expression:

```
if x > 0: 1.5 elif x > 0: -2.8 else 0.1 end
```

For any value `x > 0`, this `if` expression produces `1.5`. It will never produce `-2.8`, as the `if` part is

considered before the `elif` part.

Switch expression

Similar to an `if` expression, a `switch` expression can be used to produce different values under different circumstances. Consider the following partial CIF model:

```
enum mode = OFF, ON1, ON2, ON3, DISCONNECTED;
automaton rotary_switch:
  disc mode m = OFF;
  ...
end

alg int speed = switch rotary_switch.m:
                  case ON1: 5
                  case ON2: 10
                  case ON3: 15
                  else      0
                end;
end
```

This `switch` expression takes the operation mode `rotary_switch.m` and provides the speed corresponding to that mode. If the value of `m` matches one of the cases, the `switch` expression will produce the corresponding value. For example, if `m` has value `ON2`, this `switch` expression produces `10`. If none of the cases match, the value of the `else` case is taken. For example, if `m` has value `DISCONNECTED`, this `switch` expression produces `0`. Note that multiple values can be captured by the `else` case, i.e., you do not have to cover each possible value with a separate case.

One can use different kinds of types as long as the types of the cases match the type of the `switch` expression. In the example above, `m` is of type `mode`, which is an enumeration. Hence all cases should be of type `mode` as well. Below is an example using a ranged integer in the `switch` expression:

```
alg int[0,2] num_products;

alg string display_text = switch num_products:
                            case 0: "There are no products in the buffer."
                            case 1: "There is one product in the buffer."
                            case 2: "There are two products in the buffer."
                          end;
```

One can also use an automaton as a variable of the `switch` expression, where the cases have to be locations of that automaton:

```
automaton controller:
  location accelerate: ...
  location decelerate: ...
```

```
  location steady: ...
end

alg int acceleration = switch controller:
                         case accelerate:  5
                         case decelerate: -3
                         case steady:      0
                       end;
```

Note that in this `switch` expression we omitted the `else` case, as the three cases together already cover all possible locations of automaton `controller`.

It is possible to rewrite a `switch` expression into an `if` expression (the other way around is not always possible). This rewrite can be automated using a CIF to CIF transformation.

2.6. Scalable solutions and reuse (1/2)

2.6.1. Constants

Through the use of *constants*, fixed values can be given a name. Using constants, it is easy to change certain fixed values. If the constant is used consistently throughout the model, the value needs to be changed only in one place. Constants can thus make it easier to keep the model consistent.

Consider the following CIF specification:

```
const int STEP = 2;
const int TARGET = 100;

automaton movement:
  disc int position = 0;
  event move;

  location:
    initial;
    edge move when position < TARGET do position := position + STEP;
end
```

In this example, the `movement` automaton keeps track of the `position` of an object. The object starts at position `0`. It can `move` until it reaches its target position. The target position is `100`. Rather than using `position < 100` as guard, the value `100` is stored in a constant named `TARGET`. The constant can then be used instead the value `100`. Similarly, the step size of the object is stored in a constant named `STEP`.

Constants have a name, which by contention is usually written using upper case letters. Using a constant instead of a fixed value makes it more clear what that value represents. For instance, by using `position < TARGET` rather than `position < 100`, the intention of the guard condition is more clear. Using a constant can thus enhance readability.

Another benefit of constants, is that they can be used multiple times in the same model:

```
const int STEP = 2;
const int TARGET = 100;

automaton movement:
  disc int position = 0;
  event forward, backward;

  location:
    initial;
    edge forward  when position < TARGET do position := position + STEP;
    edge backward when position > 0      do position := position - STEP;
end
```

In this modified example it is possible for the object to perform `forward` as well as `backward` movements. The step size is the same for both movements, making it possible to use the `STEP` constant in the updates of both edges. Since a constant is used, the speed of both movements can be changed by changing the value of the constant. Without using a constant, the speed would have to be changed separately for each edge.

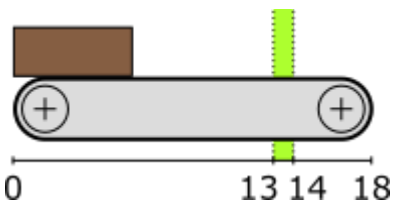
Constants are not limited to integer values. Consider the following example, where a more complex value is used:

```
enum ProductType = A, B, C;
const dict(ProductType:real) DURATION = {A: 3.5, B: 5.7, C: 0.8};
```

This example declares a `ProductType` enumeration, with three different product types: `A`, `B`, and `C`. The `DURATION` constant indicates for each product type, how long it takes to produce a product of that type. Products of type `A` can be produced in 3.5 hours, products of type `B` in 5.7 hours, etc. To get the production duration of products of type `C`, expression `DURATION[C]` can be used. For more information, see the lessons on enumerations and dictionaries.

2.6.2. Algebraic variables

Consider a conveyor belt with a product on it:



The product starts at the left side, at position `0`. There is a sensor that can detect the product between positions `13` and `14`. The product exits the conveyor at position `18`. The following CIF specification models the conveyor, product, and sensor:

```
automaton conveyor:
```

```
  disc real position = 0.0;
  event move;

  const real width = 6;
  alg bool sensor = position + width >= 13 and position <= 14;

  location:
    initial;
    edge move when position < 18 do position := position + 0.1;
end
```

The `conveyor` automaton models the conveyor, with a product on it. The product is modeled by means of the `position` of the left side of the product, relative to the left side of the conveyor. As the conveyor starts to `move`, the product moves as well, and its position on the conveyor is updated. The product moves in steps of `0.1`.

The `width` of the product is `6`. The sensor is on whenever the product, which spans from `position` to `position + width`, is within the sensor range, which spans from `13` to `14`. An *algebraic variable* named `sensor` is used here, to represent the value of the sensor.

An algebraic variable is a variable whose value is determined by its definition. For the `sensor` variable, its value is determined from a calculation involving variable `position` and constant `width`. Unlike discrete variables, algebraic variables can not be assigned a new value. The value of algebraic variable `sensor` changes automatically as the value of discrete variable `position` changes. The value of algebraic variable `sensor` is `true` whenever the product is over the sensor, and it is `false` otherwise.

If we had modeled the value of the sensor as a discrete variable, we would have had to update the variable for every edge where the value of variable `position` is updated. In this example, that is only one edge. However, if the variable would have been updated on multiple edges, the sensor value would also have to be updated for all those edges. Furthermore, when adding another edge that updates the `position` variable, the edge needs to be adapted to also update the `sensor` discrete variable, which can easily be forgotten. Using an algebraic variable, the value computation needs to be specified only once, and no changes to its value are needed, as the value always remains consistent with its definition.

Algebraic variables can be used to give an expression (computations) a name, similar to how constants can be used to give fixed values a name. The benefits of using an algebraic variable are similar to the benefits of using constants. Both can be used to improve readability, and to make it easier to consistently change the model.

Algebraic variables can also be used as an abstraction. Consider the following extension of the specification:

```
automaton light:
  event turn_on, turn_off;

  location off:
    initial;
```

```
    edge turn_on  when     conveyor.sensor goto on;

  location on:
    edge turn_off when not conveyor.sensor goto off;
end
```

The idea is to have a light turn on when a product is detected by the sensor, and have it turn off when the sensor no longer detects the product. The algebraic variable `sensor` is used in the guard conditions of the `light` automaton, to determine when the light should be turned on or off.

In the example, the `light` automaton only uses the `sensor` variable from automaton `conveyor`. It does not matter how the value of that variable is defined. Currently, it is defined in terms of variable `position` and constant `width`. However, if the `conveyor` automaton were modeled differently, the expression that defines the value of the algebraic variable could be changed, without the need to change the use of the variable in automaton `light`.

2.6.3. Algebraic variables and equations

Consider the following CIF specification:

```
automaton car:
  event start, stop, breakdown, start_repair, repaired;

  alg bool can_drive = idle or moving;

  location idle:
    initial;
    edge start goto moving;

  location moving:
    edge stop goto idle;
    edge breakdown goto broken;

  location broken:
    edge start_repair goto repairing;

  location repairing:
    edge repaired goto idle;
end
```

The `car` is initially `idle`. Once you `start` driving, the car is `moving`. Once you `stop` driving, the car is `idle` again. While `moving` it is possible for a `breakdown` to occur, meaning the car is `broken`. Once a mechanic starts the repair (`start_repair`), the mechanic is `repairing` the car. Once it is `repaired`, the car is `idle`, and you can `start` driving it again, etc.

Algebraic variable `can_drive` indicates whether you can currently drive the car. As the value calculation indicates, the car can be driven whenever it is `idle` or `moving`. That is, it can't be driven if the car is `broken` or a mechanic is `repairing` it.

In the example above, the value of the algebraic variable is defined with the declaration, as was already explained in the lesson that introduced algebraic variables. However, it also possible to specify the value separately, using an equation:

```
automaton car:
  event start, stop, breakdown, start_repair, repaired;

  alg bool can_drive;

  equation can_drive = idle or moving;

  // Locations omitted for brevity.
end
```

This allows for separation of variable declarations and equations. Both variants have the same algebraic variable with the same value. An equation of an algebraic variable must be placed in the same component as where the algebraic variable is declared. In the example above, the equation for algebraic variable `can_drive` must be placed in automaton `car`, as that is where the algebraic variable is declared.

For algebraic variables declared in automata, it is also possible to specify the value using an equation per location of the automaton:

```
automaton car:
  event start, stop, breakdown, start_repair, repaired;

  alg bool can_drive;

  location idle:
    initial;
    equation can_drive = true;

    edge start goto moving;

  location moving:
    equation can_drive = true;

    edge stop goto idle;
    edge breakdown goto broken;

  location broken:
    equation can_drive = false;

    edge start_repair goto repairing;

  location repairing:
    equation can_drive = false;

    edge repaired goto idle;
```

```
end
```

Every algebraic variable must have a unique value in every situation. Algebraic variables must thus have a value with their declaration, a single equation in the same component, or an equation in every location of the automaton. For every algebraic variable, one of the three variants must be chosen. It is allowed to choose a different variant for different algebraic variables, but it is not allowed to use multiple variants for the same algebraic variable.

Which variant fits best for a specific algebraic variable, depends on the situation. One of the benefits of using an equation per location, is that the equations are checked for completeness. If you add a new location, you must add an equation to that location as well, as otherwise the model is invalid (incomplete). This means you can't forget to specify the value of the algebraic variable for that new location. If you use a value with the declaration or a single equation in the component, you might forget to update the value for the changes you made to the automaton.

2.6.4. Type declarations

Consider this slightly modified version of an example from the lesson on constants:

```
enum ProductType = A, B, C;

const dict(ProductType:real) M1_DURATION = {A: 3.5, B: 5.7, C: 0.8};
const dict(ProductType:real) M2_DURATION = {A: 1.8, B: 4.2, C: 3.9};
```

This example declares a `ProductType` enumeration, with three different product types: `A`, `B`, and `C`. The `M1_DURATION` constant indicates for each product type, how long it takes to produce a product of that type, on machine 1. Products of type `A` can be produced in 3.5 hours, products of type `B` in 5.7 hours, etc. Constant `M2_DURATION` is similar, but for machine 2.

The type of both constants is the same. To avoid having to repeat complex types in multiple places, a *type declaration* can be used:

```
enum ProductType = A, B, C;
type Durations = dict(ProductType:real)

const Durations M1_DURATION = {A: 3.5, B: 5.7, C: 0.8};
const Durations M2_DURATION = {A: 1.8, B: 4.2, C: 3.9};
```

A type declaration with name `Durations` is introduced, and `Durations` can then be used wherever a type is expected, instead of `dict(ProductType:real)`. In the example above, `Durations` is used as type of the two constants. The original specification and the one with the type declaration have the same constants, with effectively the same type. That is, in both specification the value of constant `M1_DURATION` is a dictionary with three key/value pairs.

Type declarations can be used to give a type a name, similar to how constants can be used to give fixed values a name, and algebraic variables can be used to give computations a name. The benefits are also similar, as type declarations can be used to make specifications more concise, to increase

readability, and to make it easier to consistently change types throughout the specification.

2.7. Time

2.7.1. Timing

So far, the tutorial has only used discrete event models as examples, which are all untimed. This lesson introduces the concept of timing.

In CIF, time starts at zero (`0.0`). Time can progress continuously. That is, after one unit of time has passed, the model time is `1.0`. After an additional one and a half time units have passed, the model time is `2.5`, etc. By default, one time unit corresponds to one second. However, you can decide to use another unit, and tools such as the simulator can be configured to speed up or slow down the simulation accordingly.

Variable `time`

A variable named `time` is always available in every specification. The variable holds the current absolute model time as its value, and can be used throughout the model. Initially, time and thus the value of variable `time` start at zero (`0.0`). As time progresses, the value of variable `time` is automatically updated to ensure it properly represents the current time of the system.

In this lesson, absolute time will be used. In most models, it is easier to use relative time. This can be achieved with continuous variables, discussed in the next lesson.

Timed guards

Consider the following CIF specification:

```
event push, release;

automaton user:
  location start1:
    initial;
    edge push    when time >= 1.5 goto stop1;

  location stop1:
    edge release when time >= 2.3 goto start2;

  location start2:
    edge push    when time >= 2.4 goto stop2;

  location stop2:
    edge release when time >= 7.6 goto done;

  location done;
end
```

The `push` and `release` events represent pushing and releasing of a button respectively. The actual behavior of the button itself is omitted. The specification does model the behavior of a `user`. Initially, the `user` is in location `start1`, and no time has passed. The edge with the `push` event is not yet enabled, as the guard is not satisfied. As soon as one and a half time units have passed, the guard condition becomes satisfied, and the `push` event becomes enabled. This edge models that the user starts to push the button after `1.5` time units. The user then waits for another `0.8` (`2.3` - `1.5`) time units, before releasing the button (stop pushing it). After waiting another `0.1` (`2.4` - `2.3`) time unit, the user pushes the button again. Finally, after waiting `5.2` (`7.6` - `2.4`) time units, the user releases the button one last time. In the `done` location, the `push` and `release` events are never enabled (no edges for those events), and thus the user never pushes or releases the button again. No other events are enabled, so time keeps progressing forever, without any events happening.

Time transitions

The state space of the above specification is:



The states are labeled with the names of the current locations of automaton `user` and the current values of variable `time`. The transitions labeled with event names are *event transitions*. The other transitions are *time transitions*, which are labeled with the duration of the time transitions, i.e. the number of time units that passes. At the end of the state space, a time transition of infinite duration is shown, to indicate that time can progress forever.

The current locations of automata can not change as time passes as the result of taking a time transition. The only way for the current locations to change, is as the result of taking an edge as part of an event transition.

Urgency

By default, all events in CIF are *urgent*. Events being urgent means that edges are taken as soon as possible. In other words, event transitions take priority over time transitions. Time can only progress if no event transitions are possible. For further details on urgency, see the future urgency lesson.

Numeric time

In the above example, guard `time >= 1.5` is used. You might wonder why the guard is not `time = 1.5`, as the intention is that user pushes the button after exactly `1.5` time units, and not after `1.6` or `1.7` time units. The main reason is that the simulator uses finite precision in its numeric calculations to find the moment in time that the edge becomes enabled. The answer also has finite precision. It is often not exactly at `1.5` time units, but is slightly after it, say at time `1.50000000000001`. If you use `time = 1.5` as guard instead of `time >= 1.5`, the simulator will most likely miss the change in enabledness of the edge, and will never enable the event.

2.7.2. Continuous variables

In the lesson that introduced timing, variable `time` was used. Variable `time` uses absolute model

time, i.e. the total amount of time that has passed since the start of the simulation. It is usually easier to use relative model time, i.e. a certain amount of time passes after a certain event. This is where *continuous variables* are ideal. A continuous variable is a variable that changes value automatically, as time progresses. Consider the following CIF specification:

```
automaton machine:
  event start, finished;
  cont t = 0 der 1;

  location idle:
    initial;
    edge start do t := 0 goto producing;

  location producing:
    edge finished when t >= 3 goto idle;
end
```

This specification models a `machine` that is initially `idle`. The machine can `start` to produce a product. After a while, it is done `producing`. Due to having `finished` the product, it becomes `idle` again, until it is starts to product the next product.

Continuous variable `t` is declared to initially have value `0`. Its *derivative* is `1`, meaning that every unit of time that passes, the value of `t` increases by `1`. Every time the `start` event happens, the value of continuous variable `t` is reset to `0` using an assignment. As a result of this reset, `t` will be `0` when the automaton enters the `producing` location. The edge for the `finished` event indicates that the event can only happen when `t >= 3` holds. This condition will hold after three time units. This means that automaton `machine` remains in the `producing` location for three time units, before going to the `idle` location. It will thus always take three units after entering the `producing` location, before the guard becomes enabled, and the `finished` event can take place. The state space is as follows:

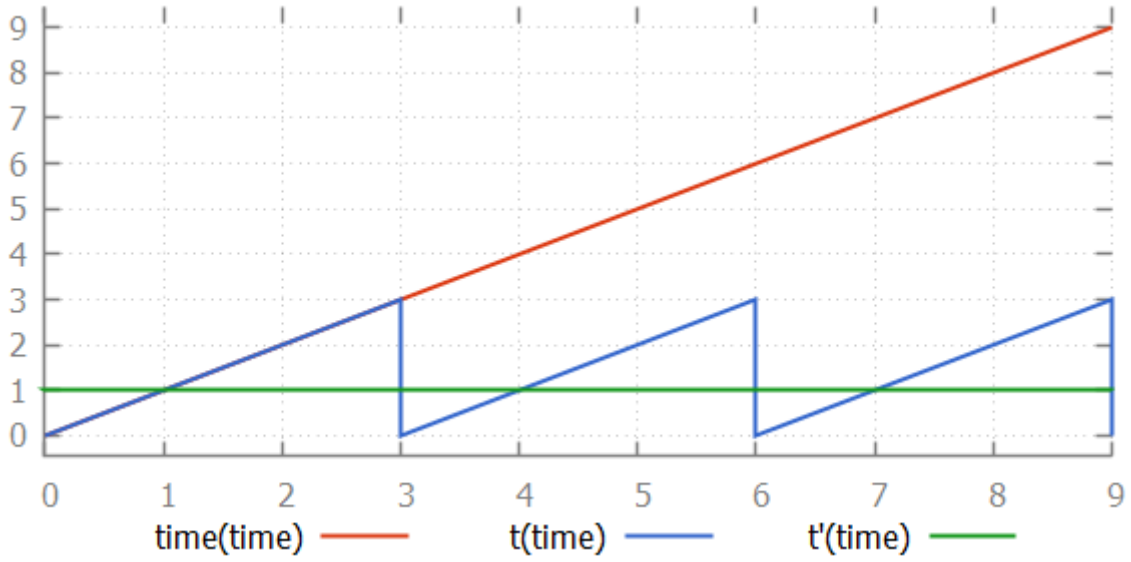


The states are labeled with the first letters of the names of the current locations of automaton `machine` and the current values of variables `time` and `t`.

Continuous variables always have real values. Similar to discrete variables, if their initial value is not specified, it is `0.0`:

```
cont t der 1; // Initial value is 0.0.
```

The derivative of a continuous variable can be used as a variable as well. The derivative of continuous variable `t` is `t'`. A derivative is read only; it can not be assigned. Similar to algebraic variables, it is always equal to its definition. In the case of variable `t`, its derivative is always `1`. The values of variables `time`, `t`, and `t'` as time progresses are:



2.7.3. Continuous variables and equations

Consider the following CIF specification:

```
automaton person:
  event turn;
  cont t = 0 der 1;

  location:
    initial;
    edge turn when t > 10 do t := 0.0;
end
```

This specification models a `person` walking back and forth. Every time that the person has walked `10` time units, (s)he will `turn` around, walking in the other direction.

In the example above, the derivative of the continuous variable is defined with the declaration, as was already explained in the lesson that introduced continuous variables. However, it also possible to specify the derivative separately, using an equation:

```
automaton person:
  event turn;
  cont t = 0;         // Declaration specifies only the initial value.

  equation t' = 1;    // Derivative specified using separate equation.

  location:
    initial;
    edge turn when t > 10 do t := 0.0;
end
```

This allows for separation of variable declarations and equations. Both variants have the same

continuous variable with the same derivative. An equation of a derivative of a continuous variable must be placed in the same component as where the continuous variable is declared. In the example above, the equation for `t'` must be placed in automaton `person`, as that is where `t` is declared.

We could extend this specification to keep track of the direction that the person is moving:

```
automaton person:
  event turn;
  cont t = 0 der 1;

  location away:
    initial;
    edge turn when t > 10 do t := 0.0 goto back;

  location back:
    edge turn when t > 10 do t := 0.0 goto away;
end
```

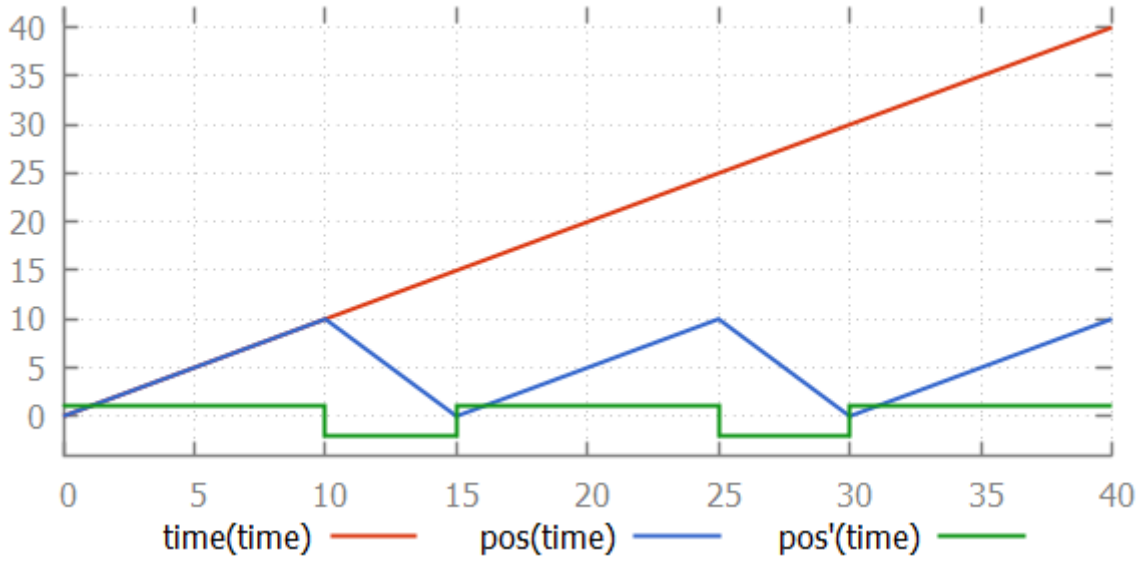
For continuous variables declared in automata, it is also possible to specify the derivative using an equation per location of the automaton. This allows us to keep track of the exact position of the person:

```
automaton person:
  event turn;
  cont pos = 0;

  location away:
    initial;
    equation pos' = 1;
    edge turn when pos > 10 goto back;

  location back:
    equation pos' = -2;
    edge turn when pos < 0 goto away;
end
```

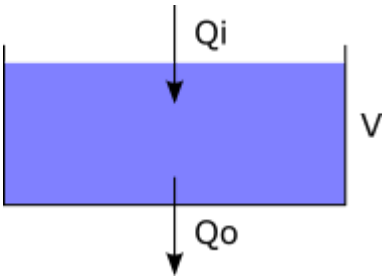
Here, the continuous variable `t` has been renamed to `pos`, to make it more clear that it indicates the position of the person. As long as the person is moving `away`, the derivative of `pos` is `1`, and the person moves away, one place every time unit. When the person reaches position `10`, the position is not reset to zero. Instead, only the location is changed to the `back` location. In that location, the derivative of `pos` is `-2`. This means that every time unit, the position decreases by `2`. That is, the person back to the original position, but at twice the speed. The values of variables `time`, `pos`, and `pos'` as time progresses are:



As with algebraic variables, every continuous variable must have a unique derivative in every situation. Continuous variables must thus have a derivative with their declaration, a single equation in the same component, or an equation in every location of the automaton. For every continuous variable, one of the three variants must be chosen. It is allowed to choose a different variant for different continuous variables, but it is not allowed to use multiple variants for the same continuous variable.

2.7.4. Equations

Consider the following non-linear system:



Variable `V` models the water volume of a tank. The water volume of the tank changes based on the incoming flow of water `Qi` and outgoing flow of water `Qo`. The tank can be modeled using the following CIF specification:

```
cont V = 5 der Qi - Qo;
alg real Qi = 1;
alg real Qo = sqrt(V);
```

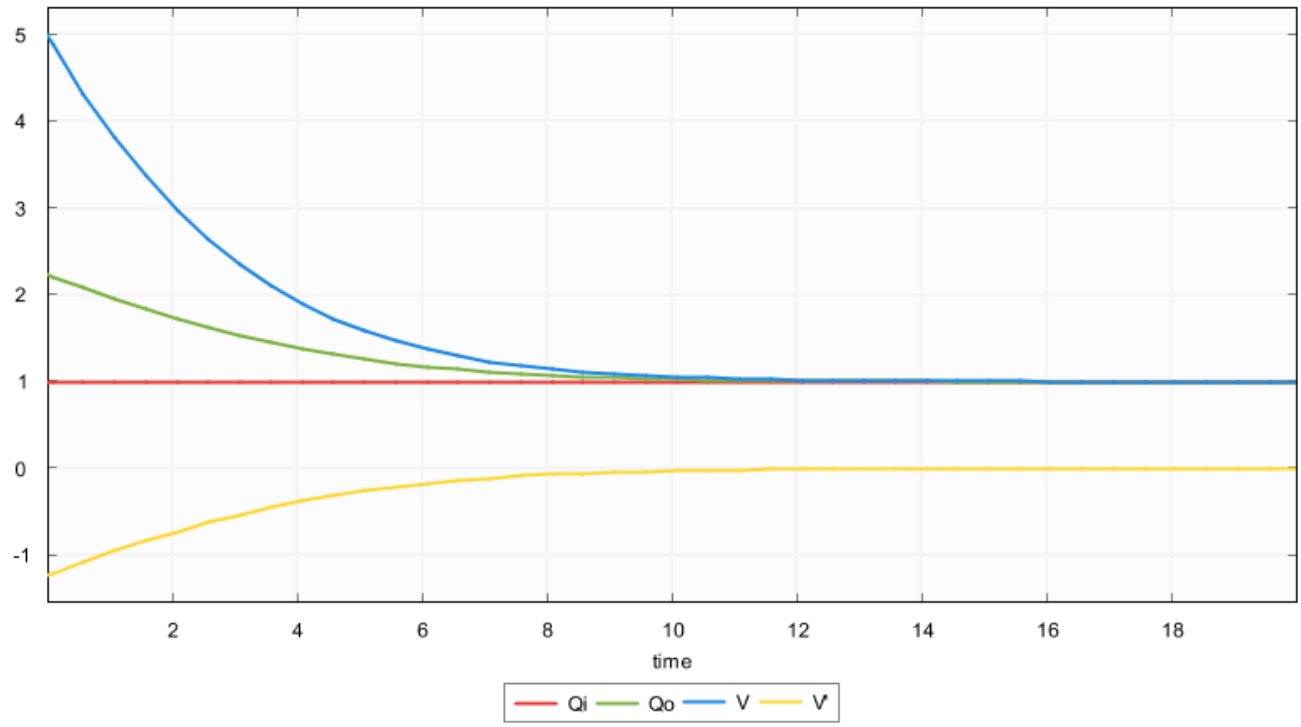
Initially, the tank is filled with `5` liters of water. The incoming flow is constant at `1` liter per second. The outgoing flow increases as the water volume of the tank increases.

By separating the declarations of the three variables from their equations, the specification becomes more readable:

```
cont V = 5;
alg real Qi;
alg real Qo;

equation V' = Qi - Qo;
equation Qi = 1;
equation Qo = sqrt(V);
```

The values of the variables as time passes are shown in the following figure:



2.7.5. Variables overview

CIF features different kinds of variables, that have different functionality. Each is better suited for a particular purpose. The following table gives an overview:

Kind of variable	Can be assigned	Can change as time progresses
Discrete variable	**yes**	no
Algebraic variable	no	**yes**
Continuous variable	**yes**	**yes**
Derivative of a continuous variable	no	**yes**
Input variable	depends	depends

Discrete variables can only change value when given an explicit new value by means of an assignment. They don't change automatically as time progresses.

Algebraic variables can't be assigned. Instead, their values depend on their declarations or equation(s). If the computations that result in their values depend on a variable that can change value as time progresses, then so can the values of the algebraic variables. Algebraic variables are used as named shorthand notations for computations, for readability, reuse, consistency, and abstraction.

Continuous variables automatically change value as time progresses, as specified by their derivatives. They can also be assigned new values, from which they then automatically change again, based on the current values of their derivatives.

The derivatives of the continuous variables can be seen as variables themselves. Their values depend on their declaration or equation(s). If the computations that result in their values depend on a variable that can change value as time progresses, then so can the values of the derivatives.

Input variables are introduced in a later lesson. Different tools may use different approaches to connect input variables to the environment of the specification, which impacts how values are provided for the input variables. Whether input variables can change value by assignment or can change as time progresses therefore also depends on how tools work with them. The CIF language allows assigning them via SVG input mappings with updates, which are explained in a later lesson.

2.7.6. Urgency

In CIF, urgency indicates whether or not time may progress. CIF has three forms of urgency: event urgency, location urgency, and edge urgency. Edge urgency should be avoided, and is not explained here.

Event urgency

By default, all events in CIF are *urgent*. Events being urgent means that edges are taken as soon as possible. In other words, event transitions take priority over time transitions. Time can only progress if no event transitions are possible. It is not possible to make events non-urgent in a CIF model, but specific tools may allow making events non-urgent.

Location urgency

Consider the following CIF specification:

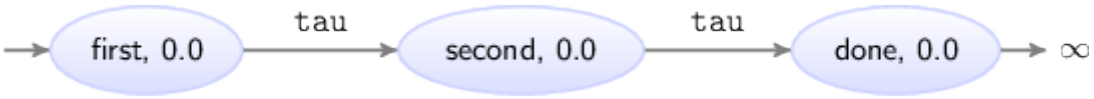
```
automaton use_case:
  location first:
    initial;
    edge tau goto second;

  location second:
    edge tau goto done;

  location done;
```

```
end
```

This automaton represents a use case, which goes from one location to the next. The state space is:



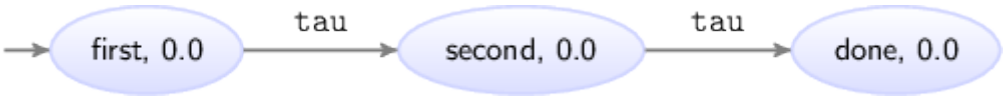
The states are labeled with the names of the current locations of automaton `use_case` and the current values of variable time. At the end of the use case, no events are possible, and thus time can progress infinitely. To force that in location `done` no time can progress, the location can be made urgent:

```
automaton use_case:
  location first:
    initial;
    edge tau goto second;

  location second:
    edge tau goto done;

  location done:
    urgent; // Location is now urgent.
end
```

The resulting state space is:



2.7.7. Deadlock and livelock

This lesson explains the concepts of deadlock and livelock.

Deadlock

If no event transitions are possible, and also no time transitions are possible (time may not progress), then no transitions are possible. This is called *deadlock*. No behavior is possible, or will ever become possible. As an example, consider the following CIF specification:

```
automaton use_case:
  location wait3:
    initial;
    urgent;
    edge tau when time > 3 goto done;

  location done;
```

```
end
```

Location `wait3` is the initial location. Initially, `time` is zero. Time can not progress, as the location is urgent. Since the guard of the edge also does not hold, event `tau` is also not possible. No transitions are possible, and the specification is in deadlock.

Livelock

As long as an event is possible, no time may pass. If by mistake always an event is possible, this prevents all passage of time. Consider the following CIF specification:

```
automaton car:
  event increase, decrease, arrived;
  disc real speed = 0;
  cont pos = 0 der speed;

  location underway:
    initial;
    edge increase when speed < 100 do speed := speed + 1;
    edge decrease when speed > 0   do speed := speed - 1;
    edge arrived  when pos = 850 goto done;

  location done;
end
```

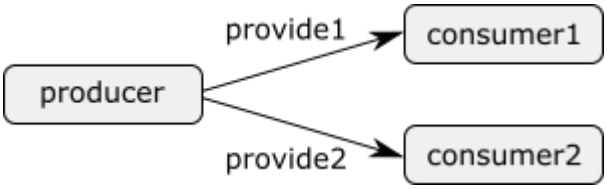
This specification models a `car` that can `increase` and `decrease` its speed, so that it is at least `0` and at most `100`. It keeps track of its position (continuous variable `pos`) that increases more quickly as the speed increases. When position `850` is reached, the car has `arrived` and is `done`.

The car will be increasing and decreasing it speed, using the `increase` and `decrease` events. As always at least one of them is enabled at any moment, time can never progress, and `pos` remains zero. This effect is called *livelock*. While behavior is still possible, the model does not truly progress with useful behavior. The model gets 'stuck' repeating one or a few events.

2.8. Channel communication

2.8.1. Channels

Consider the following figure of a producer and two consumers, where rectangles represent entities and the arrows represent the data that is communicated:



The producer creates products, identified by a unique identification number. Each product produced by the producer, is provided either to the first or to the second consumer. Consider the following CIF specification:

```
event provide1, provide2;

automaton producer:
  disc int nr = 0;

  location:
    initial;
    edge provide1, provide2 do nr := nr + 1;
end

automaton consumer1:
  disc list int nrs;

  location:
    initial;
    edge provide1 do nrs := nrs + [producer.nr];
end

automaton consumer2:
  disc list int nrs;

  location:
    initial;
    edge provide2 do nrs := nrs + [producer.nr];
end
```

The `producer` keeps track of the identification number (variable `nr`) of the current product, and provides products to either the first consumer (event `provide1`) or the second consumer (`provide2`). Both consumers have a list of their products (variable `nrs` in the consumer automata). Initially, the consumers don't have a product, and the list is empty. When a consumer gets a new product, it looks up the identification number of the product at the producer, and stores it locally. The producer then moves on to the next product, by increasing its current identification number.

We can identify two problems in this model.

The first problem is that we need two events in order for the producer to provide products to *either* the one consumer *or* the other consumer. If we used only one event, both consumers would need to use that event, have the event in their alphabet, and would thus have to simultaneously participate in the synchronization. A consequence of having an event per consumer, is that the `producer` automaton has both events on its edge. Adding a third consumer entails having to add another event, as well as having to modify the edge of the `producer` automaton. This is not a nice scalable solution.

The second problem is that the consumer refers directly to the `nr` variable of the `producer` automaton. This introduces a very tight coupling between the producer and the consumers. It

exposes the `nr` variable of the `producer` to the consumers, making it more difficult to change the producer without changing the consumers.

Both these problems can be solved by using *channels*. Channels are a special form of events, that can be used to communicate or transmit data from a *sender* to a *receiver*. In our example, data that is communicated are the identification numbers of the products, the producer is the sender, and the consumers are the receivers.

Channels require one or more potential senders, and one or more potential receivers. Automata cannot be both sender and receiver for a single channel. They may however be a sender for one channel, and a receiver for another channel. For every transition, exactly one of the senders and exactly one of the receivers participate. The sender sends a value, and the receiver receives that value. This type of communication is often called channel communication or *point-to-point* communication, as the data is communicated from one point (the sender) to another point (the receiver).

Multiple automata that synchronize over the same event perform a transition together. Similarly, a sender and receiver that together perform a channel communication, perform a transition together. In both cases, all automata involved take their respective edges *synchronously* (simultaneously).

Channels are ideally suited for modeling product flows, or more generally the movement of physical entities through a system. Physical objects usually don't duplicate themselves or spontaneously stop to exist. This fits nicely with channels, where data is communicated or passed along from exactly one sender to one receiver. In our example, product produced by the producer are physically provided to one of the consumers.

The following CIF specification models the above example using channels:

```
event int provide;

automaton producer:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton consumer1:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end

automaton consumer2:
  disc list int nrs;
```

```
  location:
    initial;
    edge provide? do nrs := nrs + [?];
end
```

The `provide1` and `provide2` events have been replaced by a single channel named `provide`. Channels are declared similar to events, but have a data type that indicates the type of values that are communicated over the channel. In this case integers are communicated.

The `producer` now uses the channel on its edge, instead of the two events. The exclamation mark (`!`) after the channel name means that the producer is sending over the channel. After the exclamation mark, the value that the producer sends is given. In this case, the producer sends the identification number of its current product.

The edges of the consumers have been modified as well. The channel is used with a question mark (`?`) after the channel name, indicating that the consumers receive over the channel. The received value, which is available as the `?` variable in the update, is directly added to the `nrs` list of the consumer.

By using channels, we no longer need multiple events, and the producer does not need to be modified if another consumer is added. This makes the model scalable to varying amount of consumers. Furthermore, the consumers now use the `?` variable to obtain the received value, and no longer need direct access to the variables of the producer. This makes it easier to modify the producer without having to also modify the consumers.

To conclude this lesson, we'll extend the example with a second producer:

```
event int provide;

automaton producer1:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton producer2:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton consumer1:
  disc list int nrs;

  location:
```

```
    initial;
    edge provide? do nrs := nrs + [?];
end

automaton consumer2:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end
```

The `producer` automaton has been renamed to `producer1`, and a `producer2` has been added. Both producers independently produce products and provide them to the consumers. Both consumers can receive products from either producer. At all times, four transitions are possible: `producer1` communicates with `consumer1`, `producer1` communicates with `consumer2`, `producer2` communicates with `consumer1`, or `producer2` communicates with `consumer2`.

Note that the `producer1` and `producer2` automata are identical, as are the `consumer1` and `consumer2` automata. In the lesson on automaton definition/instantiation, it is shown how this duplication can be prevented.

2.8.2. Dataless channels

Besides channels that communicate data, it is also possible to use dataless channels. A channel that does not communicate any data, is declared with the `void` type. Dataless channels are also called `void` channels. When using dataless channels, the sender does not provide a value to send over the channel. Also, the receiver cannot use the received value (variable `?`) since no data is communicated.

Dataless channels are used in the same situations as 'normal' channels, for instance when physical product flow is modeled. If products don't have an identification number, and can not be distinguished based on color or some other property, all products are essentially equal. It is then sufficient to communicate that a product is being 'transferred'. No further data is needed. While it is possible to communicate dummy values, dataless channels provide a better solution in such cases.

Consider again the producer/consumer example from the previous lesson, with one producer and two consumers, but without identification numbers for the products:

```
event void provide;

automaton producer:
  location:
    initial;
    edge provide!;
end

automaton consumer1:
```

```
  location:
    initial;
    edge provide?;
end

automaton consumer2:
  location:
    initial;
    edge provide?;
end
```

The `producer` still produces products and provides them to either the first or the second consumer.

2.8.3. Combining channel communication with event synchronization

Consider again the producer/consumer example with two producers and two consumers, from a previous lesson:

```
event int provide;

automaton producer1:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton producer2:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton consumer1:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end

automaton consumer2:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
```

```
end
```

Now assume we want to restrict communication to allow at most five products in total to be provided to the consumers. We could adapt both producers, as follows:

```
automaton producer1:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr when nr + producer2.nr < 5 do nr := nr + 1;
end

automaton producer2:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr when producer1.nr + nr < 5  do nr := nr + 1;
end
```

Each producer gets an additional guard condition whether it may provide a product to one of the consumers. If the total number of products provided by both providers does not exceed five, they may still provide a product. Having to adapt both producers is less than ideal.

As an alternative solution, we can add an additional automaton, instead of adapting the producers:

```
automaton controller:
  disc int cnt = 0;

  location:
    initial;
    edge provide when cnt < 5 do cnt := cnt + 1;
end
```

This `controller` automaton (together with the original producer automata) keeps track of the number of products provided to consumers, by counting them in variable `cnt`. It only allows the `provide` event when less than five products have been provided. If five or more products have been provided, it disables the `provide` event.

In the `controller` automaton, the `provide` channel is used as an event rather than a channel. When one of the producers and one of the consumers together perform a channel communication, the `controller` automaton that has the `provide` event in its alphabet, must synchronize with it. This allows the `controller` to impose additional restrictions on the channel communication, allowing or forbidding it in certain cases. The `controller` is added as a separate process, which improves scalability.

In general, every automaton may either send over a channel, receive over a channel, or

synchronize with a channel. An automaton may not take on more than one of these roles, for a single event. It may however send over one channel, receive over another channel, and synchronize with yet another one.

Every event transition for a channel requires exactly one automaton that participates as sender, and exactly one automaton that participates as receiver. Furthermore, all automata that have the channel in their alphabet, must additionally participate as well, by synchronizing together with the sender and receiver. Automata that send or receive over a channel, do not have that channel in their alphabet. Only automata that synchronize with an event or channel have that event or channel in their alphabet.

Automata that synchronize over a channel can be used to further *restrict* the allowed channel communications, as shown in the above example. It is however also possible for the additional synchronizing automata to monitor (observe) the channel communication.

2.9. Functions

2.9.1. Functions

Functions can be used to compute values from other values. CIF has many built-in functions, called *standard library functions*. An example is the `size` function, which takes a list and returns the number of elements of the list. Another example is the `abs` function, which takes an integer or real number and computes the absolute value of that number.

While many built-in functions are available, they may not always suffice. User-defined functions can be added to CIF models, to allow custom computations needed by the model. By putting the calculation in a user-defined function, the calculation can be used in several places, allowing for reuse. Furthermore, functions allow for more complex computations than would otherwise be possible.

CIF features two kinds of user-defined functions: internal ones and external ones. Internal user-defined functions are fully defined within the CIF model. External user-defined functions declare only a header which indicates the existence of the function, while the actual implementation is obtained from an external source. The lessons of this category only explain internal user-defined functions. The language reference documentation gives more information on external user-defined functions.

2.9.2. Internal user-defined functions

The most commonly used kind of user-defined functions are the internal user-defined functions. Consider the following CIF specification:

```
func real mean(list real vs):
  int length = size(vs);
  int index = 0;
  real sum = 0;

  while index < length:
```

```
    sum := sum + vs[index];
    index := index + 1;
  end
  return sum / length;
end
```

This specification defines a function (keyword `func`) named `mean`. After the name of the function, between the parentheses, the *parameters* are listed. The parameters are the input values, which the function can use to compute its result. In this example, the function takes a list of real values as its only parameter. Parameter name `vs` can be used in the body of the function to refer to this input value. Just after the `func` keyword, the type of the *result* of the computation is specified. In this case, the function results in a real typed value. The `mean` function takes a list of integer values and produces a single real value as result.

In the *body* of the function, local variables can be declared. The `mean` function declares three variables: `length`, `index`, and `sum`. Local variables of functions are also called function variables, and they are very similar to discrete variables. The main difference is that they are declared without the `disc` keyword. In the example, `length` is set to the number of elements in list `vs`. Variables `index` and `sum` are both initialized to `0`.

After the local variables (if any), the statements of the body are given. The statements implement the algorithm, the actual computation of the function. Statements are executed one after another, in the order they are given. In the `mean` function, the `while` statement is executed before the `return` statement. The `mean` function first calculates the sum of the input values, and then *returns* the mean value. The details of the statements are discussed in the next lesson.

A function can be called (or applied) on concrete input values, to obtain the computation result for those specific input values. For instance, consider the following extension to the above CIF specification:

```
alg real m = mean([1.5, 3.2, 7.9, 15.8]);

automaton a:
  disc real x;

  location:
    initial;
    edge do x := 2 * mean([0.4, 1.5, 6.8]);
end
```

Algebraic variable `m` is given the value that results from calling function `mean` on a single *argument*, a list with four values. Each argument of a function call must match with the corresponding parameter of the function being called. In this case, the list of four real values matches with the `vs` parameter of the `mean` function, which has type `list real`. Variable `m` becomes `7.1`, as the mean of those four values is `7.1` (`(1.5 + 3.2 + 7.9 + 15.8) / 4`).

The edge in automaton `a` assigns a value to variable `x`. The mean of a list of three values is calculated, and multiplied by two, to obtain the new value of `x`. The mean of `0.4`, `1.5`, and `6.8` is `(0.4`

`+ 1.5 + 6.8) / 3`, which is `2.9`. Variable `x` gets `5.8` (`2.9 * 2`) as its new value.

Function `mean` is called in two places in the example, showing reuse of calculations.

Functions in CIF are mathematical functions. That is, the result of a function is the same for the same input values, and functions have no side effects. Functions can not directly access variables outside their body. For example, they cannot access discrete, continuous, and algebraic variables. They can also not use variable `time`. To use the values of those variables in a function, they have to be passed in through parameters.

2.9.3. Function statements

This lesson explains the different statements that can be used in internal user-defined functions:

- Assignment statement
- Return statement
- While statement
- If statement
- Break statement
- Continue statement

Assignment statement

Assignments in functions are very similar to assignments to discrete variables. For instance:

```
x := x + 1;
```

Besides local variables, it is also allowed to assign new values to the parameters. Changes to parameters only have effect inside the function itself.

For multi-assignments, the outer parentheses are not needed:

```
// Multi-assignment on an edge.
edge do (x, y) := (1, 2);

// Multi-assignment in a function.
x, y := 1, 2;
```

Return statement

The execution of a function ends when a `return` statement is encountered. The value following the `return` statement is the result of the entire function. Consider again the `return` statement from the

`mean` function from the previous lesson:

```
return sum / length;
```

In this case, the `sum` of the input values is divided by the number of input elements (variable `length`) to obtain the mean of the input values. The mean of the input values is the result of the function.

While statement

The `while` statements allows for repeated execution of the statements in its body, as long as the condition of the `while` loop holds. Consider again the `mean` function from the previous lesson:

```
func real mean(list real vs):
  int length = size(vs);
  int index = 0;
  real sum = 0;

  while index < length:
    sum := sum + vs[index];
    index := index + 1;
  end
  return sum / length;
end
```

The `mean` function processes each of its input values, using a `while` loop. As long as the condition (`index < length`) holds, the body of the `while` is executed over an over. The body consists of two assignments. The first assignment obtains an input value (`vs[index]`) and adds it to the `sum`. The second statement increases the index, to proceed with the next input value. After the two assignments have been executed, the condition of the `while` is evaluated again. If the condition still holds, the two assignments are executed again, etc. When the condition no longer holds, the `while` statement is done, and execution proceeds with the next statement, the `return` statement.

If statement

In a previous lesson, the `if` expression is introduced to construct more complex expressions. Consider the `if` expression from that previous lesson, but now used within a function:

```
func int signum(real x):
  return if   x > 0:  1
         elif x < 0: -1
         else         0
         end;
end
```

For functions one can also use `if` statements. This expression can be rewritten into an `if` statement as follows:

```
func int signum(real x):
  if x > 0:
    return 1;
  elif x < 0:
    return -1;
  else
    return 0;
  end
end
```

A difference between an `if` statement and an `if` expression is that the `if` statement can be partial in the sense that it does not need an `else` part. Consider again the `mean` function from above. It does not correctly handle empty lists. The length of an empty list is `0`, so the `return` statement will cause a division-by-zero error. We can correctly handle this edge case using an `if` statement:

```
func real mean(list real vs):
  int length = size(vs);
  int index = 0;
  real sum = 0;

  if length = 0:
    return 0;
  end

  while index < length:
    sum := sum + vs[index];
    index := index + 1;
  end
  return sum / length;
end
```

Break statement

A `break` statement 'jumps out' of a `while` statement, continuing with the first statement after the `while`. Consider the following CIF specification:

```
// Get the first 'n' values from 'xs' that are not 'bad'.
func list int first_n(list int xs; int n; int bad):
  int index = 0;
  int x;
  list int result = [];

  while index < size(xs):
```

```
    x := xs[index];
    if x != bad:
      result := result + [x];
    end
    if size(result) = n:
      break;
    end
    index := index + 1;
  end
  return result;
end

// The value of 'y' is [1, 5, 3].
alg list int y = first_n([1, 2, 5, 3, 4, 1, 3], 3, 2);
```

The `first_n` function takes a list of integer numbers `xs`, and returns a list `result` with the first `n` numbers from `xs` that are not `bad`. A `while` loop is used to process each of the numbers in list `xs`. The current number is stored in variable `x`. If the current number is not the same as `bad`, it is added to the `result`. If the `result` then contains `n` values, the `break` statement is used to break out of the `while` loop and continue execution at the first statement after the `while` loop, which is the `return` statement. If less than `n` values are in the result, `index` is incremented to ensure the next number of the list is processed in the next iteration of the `while` loop. The `return` statement at the end of the function is used to return the `result` list, which contains at most `n` values.

Continue statement

A `continue` statement 'jumps over' the remaining statements in the body of a `while` statement, and continues with the next iteration of that `while` statement. Consider the following CIF specification:

```
// Get the values from 'xs' that are greater than 5.
func list int filter_gt5(list int xs):
  int index = 0;
  int x;
  list int result = [];

  while index < size(xs):
    x := xs[index];
    index := index + 1;
    if x <= 5:
      continue;
    end
    result := result + [x];
  end
  return result;
end

// The value of 'y' is [8, 7, 6].
```

```
alg list int y = filter_gt5([1, 8, 5, 7, 4, 6, 3]);
```

The `filter_gt5` function takes a list of integer numbers, and returns a filtered list that only contains those integer number that are greater than 5. A `while` loop is used to process each of the numbers in list `xs`. The current number is stored in variable `x`, and `index` is incremented to ensure the next number of the list is processed in the next iteration of the `while` loop. If the current number is not greater than 5, the remaining statements in the body of the `while` are skipped, meaning `x` is not added to the `result`. The `while` loop then continues with the next iteration, for the next number in `xs`. If the current number (`x`) is greater than 5, execution continues after the `if` statement, and the number is added to the `result`. After all numbers in the list have been processed, the filtered `result` list is returned to the caller of the function.

2.9.4. Functions as values

Functions can be used as values. By treating them as data, they can be stored in variables, and passed to other functions. This lesson shows one example of how that can be useful.

Consider the following list of numbers:

```
alg list real nrs = [0.5, 1.3, 0.1, 2.7, 1.4];
```

Now assume we wanted to sort these numbers both in increasing and in decreasing order, using a single sorting function:

```
alg list real inc = sort(nrs, cmp_inc); // [0.1, 0.5, 1.3, 1.4, 2.7]
alg list real dec = sort(nrs, cmp_dec); // [2.7, 1.4, 1.3, 0.5, 0.1]
```

Variable `inc` contains the same numbers as `nrs`, but sorted in increasing order, while `dec` contains them in decreasing order. We use the same `sort` function in both cases, but with different comparison functions:

```
func bool cmp_inc(real a, b):
  return a <= b;
end

func bool cmp_dec(real a, b):
  return a >= b;
end
```

Function `cmp_inc` takes two real numbers and returns `true` only if the first number is smaller than the second one (`a` and `b` are in increasing order). Function `cmp_dec` has the same parameters, but returns `true` only if the first number is larger than the second one (`a` and `b` are in decreasing order). The `sort` function is defined as follows:

```
func list real sort(list real xs; func bool (real, real) cmp):
```

```
  int i = 1, j;
  real x;

  while i < size(xs):
    j := i;
    while j > 0 and not cmp(xs[j-1], xs[j]):
      // swap x[j-1] and x[j]
      x := xs[j-1];
      xs[j-1] := xs[j];
      xs[j] := x;
      j := j - 1;
    end
    i := i + 1;
  end
  return xs;
end
```

The `sort` function has two parameters. The first parameter is `xs`, which contains the values to sort. The second parameter is `cmp`, the compare function to use to determine whether two numbers are correctly ordered. The `cmp` parameter has type `func bool (real, real)`, which means that a function that has two real parameters and a boolean return value is required. The `cmp_inc` and `cmp_dec` functions satisfy these requirements, and can be used as second argument when the function is called to determine the values of algebraic variables `inc` and `dec`.

The `sort` function implements a standard insertion sort algorithm. The `cmp` parameter is used in the `sort` function to compare two consecutive values in `xs`, and swap them if they are not correctly ordered.

The `cmp` parameter of the `sort` function has a function type, allowing compare functions to be passed to the `sort` function, as data. This allows the `sort` function to sort lists of numbers in different orders, depending on the compare function that is provided.

2.10. Scalable solutions and reuse (2/2)

2.10.1. Automaton definition/instantiation

Parts of a system that are nearly identical, are often modeled as nearly identical automata. Having to specify them multiple times can be burdensome. It can also hinder scalability, as changes to one of them usually need to be applied to the others as well. Consider again the producer/consumer example from the lesson that introduced channels:

```
event int provide;

automaton producer:
  disc int nr = 0;

  location:
    initial;
```

```
    edge provide!nr do nr := nr + 1;
end

automaton consumer1:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end

automaton consumer2:
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end
```

The producer provides products either to the first consumer or to the second consumer. The consumers are modeled using identical automata. Only the names of the `consumer1` and `consumer2` automata differ. Ideally, we would have a sort of consumer template, and use that template twice, once for each of the actual consumers. This can be achieved in CIF using an *automaton definition* (the template) and two *automaton instantiations* (the uses of the template):

```
event int provide;

automaton producer:
  disc int nr = 0;

  location:
    initial;
    edge provide!nr do nr := nr + 1;
end

automaton def Consumer():
  disc list int nrs;

  location:
    initial;
    edge provide? do nrs := nrs + [?];
end

consumer1: Consumer();
consumer2: Consumer();
```

The `Consumer` automaton definition is identical to the original consumers, except that it is an automaton definition rather than an automaton. An automaton definition can be identified by the `def` keyword between the `automaton` keyword and the name of the automaton definition, as well as

by the parentheses after its name. As a convention, names of automaton definitions start with an upper case letter (`Consumer` rather than `consumer`).

An automaton definition by itself is not an automaton. The instantiations of the automaton definition (`consumer1` and `consumer2`) however, *are* automata. Before the colon (`:`), the name of the instantiation is given. This name is also the name of the actual automaton. After the colon, the name of the automaton definition that is instantiated is given.

Using an automaton definition, the above example models the behavior of a consumer only once. Adding a third consumer is as easy as adding another automaton instantiation, which takes only one line of code (`consumer3: Consumer();`). Changing the behavior of all consumers only requires changes to the common automaton definition. Automaton definition/instantiation allows for scalability and reuse, and also improves maintainability.

Automaton definition/instantiation can be eliminated, by replacing all automaton instantiations by the automaton definitions that they instantiate, and changing the automaton definition header (`automaton def Consumer():`) by an automaton header (`automaton consumer1:`). If we do that for the example above, we obtain the original specification from the beginning of this lesson. The two specifications are functionally equivalent. Automaton instantiation `consumer1` is also often referred to as automaton `consumer1`, when there is no confusion.

2.10.2. Parametrized automaton definitions

In the previous lesson, automaton definition/instantiation was used to obtain two identical automata, while only having to specify their behavior once. What we have seen so far, is enough for exactly identical automata, but not for nearly identical automata. Consider the following two nearly identical consumers:

```
automaton consumer1:
  disc list int buffer = [];

  location:
    initial;
    edge provide? when size(buffer) < 2 do buffer := buffer + [?];
end

automaton consumer2:
  disc list int buffer = [];

  location:
    initial;
    edge provide? when size(buffer) < 3 do buffer := buffer + [?];
end
```

The consumers can accept products that the producer provides (channel `provide`). They store the identification numbers of those products in a `buffer`. The two consumers are identical except for the number of products that they can accept: the first consumer can accept two products, the second producer can accept three products. We can still use automaton definition and instantiation

to model the consumer only once, but we need to parametrize the automaton definition:

```
automaton def Consumer(alg int capacity):
  disc list int buffer = [];

  location:
    initial;
    edge provide? when size(buffer) < capacity do buffer := buffer + [?];
end

consumer1: Consumer(2);
consumer2: Consumer(3);
```

The `Consumer` automaton definition now has a parameter named `capacity` that indicates how many identification numbers can be stored in its `buffer`. The automaton instantiations `consumer1` and `consumer2` provide an argument (`2` and `3` respectively) to match the parameter of `Consumer`. That is, the instantiations indicate their `capacity`. Using parameters, the `Consumer` automaton definition models the behavior of both automata `consumer1` and `consumer2`, even though they have different capacities.

The details of the different kind of parameters of automaton definitions are explained in the next lesson.

2.10.3. Automaton definition parameters

In the previous lesson, an automaton definition with parameter was used. The parameter was an algebraic parameter, which is only one of the different kinds of automaton definition parameters. This lesson explains each of them:

- Algebraic parameters
- Event parameters
- Location parameters
- Input parameters
- Automaton parameters

This lesson also explains how to use multiple parameters.

Algebraic parameters

An algebraic parameter is similar to an algebraic variable. Arbitrary values or expressions of matching type can be provided as arguments in automaton instantiations. For instance, consider the following partial CIF specification:

```
event int accept, provide;
```

```
automaton def Buffer(alg int capacity):
  disc list int buf = [];

  location:
    initial;
    edge accept?        when size(buf) < capacity do buf := buf + [?];
    edge provide!buf[0] when size(buf) > 0        do buf := buf[1:];
end

buffer1: Buffer(5);
```

Automaton definition `Buffer` has an algebraic parameter that indicates the `capacity` of the buffer. The buffer can `accept` something when it has not yet reached its capacity. It can `provide` something when the buffer is not empty. Automaton instantiation `buffer1` has value `5` as its argument. Value `5` is an integer number, which matches the integer type (`int`) of the `capacity` parameter.

Algebraic parameters can be used inside an automaton definition, wherever a value is expected, e.g. in guards, updates, initial values of discrete variables, and invariants. The expression that is provided by the instantiation is essentially filled in wherever the parameter is used. The above is equivalent to:

```
event int accept, provide;

automaton buffer1:
  disc list int buf = [];

  location:
    initial;
    edge accept?        when size(buf) < 5 do buf := buf + [?];
    edge provide!buf[0] when size(buf) > 0 do buf := buf[1:];
end
```

Event parameters

Event parameters allow different instantiations to synchronize with different events or to communicate over different channels. For instance, consider the following partial CIF specification:

```
event int generate, pass_along, exit;

automaton def Buffer(event int accept, provide):
  disc int buffer;

  location accepting:
    initial;
    edge accept? do buffer := ? goto providing;
```

```
  location providing:
    edge provide!buffer goto accepting;
end

buffer1: Buffer(generate, pass_along);
buffer2: Buffer(pass_along, exit);
```

Automaton definition `Buffer` is parametrized with two channels, one to `accept` a product into the one place buffer, and one to `provide` it to some other part of the system. The first buffer (`buffer1`) accepts products via the `generate` channel, and provides products via the `pass_along` channel. The second buffer (`buffer2`) accepts products via the `pass_along` channel, and provides products via the `exit` channel. The first buffer uses the `pass_along` channel as its `provide` channel parameter, and the second buffer uses that same `pass_along` channel as its `accept` channel parameter. The first buffer thus provides its items to the second buffer.

Event and channel parameters can be used inside an automaton definition, wherever an event or channel is expected, e.g. on edges and in alphabets if explicitly specified. The event or channel that is provided by the instantiation is essentially filled in wherever the parameter is used. The above is equivalent to:

```
event int generate, pass_along, exit;

automaton buffer1:
  disc int buffer;

  location accepting:
    initial;
    edge generate? do buffer := ? goto providing;

  location providing:
    edge pass_along!buffer goto accepting;
end

automaton buffer2:
  disc int buffer;

  location accepting:
    initial;
    edge pass_along? do buffer := ? goto providing;

  location providing:
    edge exit!buffer goto accepting;
end
```

Channel parameter usage restrictions

If an event parameter is actually a channel (it has a data type), it may also be called a *channel*

parameter. By default, a channel parameter can be used to send, receive, or synchronize. However, it is also possible to restrict the allowed usages. By putting certain event usage restriction flags after the name of the channel parameter, only those usages are allowed. The available flags are `!` to allow sending, `?` to allow receiving, and `~` to allow synchronizing. Duplicate flags are not allowed, the `!` should be before the `?` flag, and the `!` and `?` flags should be before the `~` flag.

By restricting the usages, you can immediately see how a channel parameter is used within the automaton definition, as only the usages indicated by the parameter are allowed. It serves as sort of documentation of the intention of the channel parameter. This is much simpler than finding the actual usages of the parameter in the automaton. However, note that the parameter indicates the allowed usages, and doesn't guarantee that the event/parameter is actually used at all.

Another benefit of restricting the usages, is that it makes it possible to spot mistakes. You might for instance use a channel parameter on an edge (e.g. `edge e`), but forget to include the send part (e.g. `edge e!1`). If the parameter only allows sending, the accidental synchronization (`edge e`) is reported as an invalid use. Without the usage checking, you might not encounter the problem until for instance simulation, where it is much more difficult to find the cause.

For the example above, we have the following *header* of the `Buffer` definition:

```
automaton def Buffer(event int accept, provide):
```

We can change this as follows:

```
automaton def Buffer(event int accept?, provide!):
```

This makes it clearer that the `accept` channel is used to receive a product into the buffer, and the `provide` channel is used send a product from the buffer.

Location parameters

An earlier lesson explained how a location can be used as a variable. Using location parameters, automaton definitions can be supplied with different locations. For instance, consider the following CIF specification:

```
automaton def Machine(location other_processing):
  location heat_up:
    initial;
    edge when not other_processing goto processing;

  location processing:
    edge tau goto cool_down;

  location cool_down:
    edge tau goto heat_up;
```

```
end

machine1: Machine(machine2.processing);
machine2: Machine(machine1.processing);
```

Automaton definition `Machine` represents a machine that can heat up, process something, cool down, and repeat that forever. The system consists of two of those machines. The machines can not start processing if the other machine is already processing. That is, the machines perform mutually exclusive processing. If the first machine is in its `processing` location, the other can't also be processing (in its own `processing` location). To prevent a machine from starting to process if the other machine is already processing, each machine needs to know whether the other is already processing. Therefore, automaton definition `Machine` is parametrized with a location parameter `other_processing`, that indicates whether the other machine is currently processing (in its `processing` location). This parameter is used as a guard that determines whether a transition from location `heat_up` to location `processing` is allowed. Automaton instantiation `machine1` provides the first machine with the `processing` location of the second machine, by using `machine2.processing` as its instantiation argument.

Location parameters can be used inside an automaton definition, wherever a boolean value is expected, e.g. in guards. The location that is provided by the instantiation is essentially filled in wherever the parameter is used. The above is equivalent to:

```
automaton machine1:
  location heat_up:
    initial;
    edge when not machine2.processing goto processing;

  location processing:
    edge tau goto cool_down;

  location cool_down:
    edge tau goto heat_up;
end

automaton machine2:
  location heat_up:
    initial;
    edge when not machine1.processing goto processing;

  location processing:
    edge tau goto cool_down;

  location cool_down:
    edge tau goto heat_up;
end
```

Input parameters

A future lesson explains input variables. Using input parameters, automaton definitions can be supplied with different input variables, similar to how different locations can be supplied using location parameters.. For instance, consider the following CIF specification:

```
automaton def Machine(input bool start, stop):
  location off:
    initial;
    edge when start and not stop goto on;
  location on:
    edge stop goto off;
end

input bool start1, start2;
input bool stop1, stop2;

machine1: Machine(start1, stop1);
machine2: Machine(start2, stop2);
```

As for input variables, SVG input mapping can be used to update the value of input parameters.

Automaton parameters

When multiple declarations (variables, locations, etc) from one automaton are to be supplied as parameters to another automaton, it is also possible to supply the entire automaton as a parameter, but only if the provided automaton is an instantiation of an automaton definition. Consider for instance the following CIF specification:

```
automaton def Sensor():
  event go_on, go_off;

  location off:
    initial;
    edge go_on goto on;

  location on:
    edge go_off goto off;
end

sensor1: Sensor();
sensor2: Sensor();

automaton def Actuator(Sensor sensor):
  event turn_on, turn_off;

  location off:
```

```
    initial;
    edge turn_on  when sensor.on  goto on;

  location on:
    edge turn_off when sensor.off goto off;
end

actuator1: Actuator(sensor1);
```

Automaton definition `Sensor` models a sensor that can go on an off. Both `sensor1` and `sensor2` are actual sensors. Automaton definition `Actuator` models an actuator that can be turned on if a sensor is on, and be turned off if that same sensor is off. The `actuator1` automaton is provided `sensor1` as sensor. If `sensor1` goes on, `actuator1` is turned on, and if `sensor1` goes off, `actuator1` is turned off. `sensor2` going on or off has no effect on `actuator1`.

Automaton parameters can be used inside an automaton definition, to refer to declarations inside the automaton supplied for the automaton parameter. The automaton that is provided by the instantiation is essentially filled in wherever the parameter is used. The above is equivalent to:

```
automaton sensor1:
  event go_on, go_off;

  location off:
    initial;
    edge go_on goto on:

  location on:
    edge go_off goto off:
end

automaton sensor2:
  event go_on, go_off;

  location off:
    initial;
    edge go_on goto on:

  location on:
    edge go_off goto off:
end

automaton actuator1:
  event turn_on, turn_off;

  location off:
    initial;
    edge turn_on  when sensor1.on  goto on;

  location on:
```

```
    edge turn_off when sensor1.off goto off;
end
```

Multiple parameters

It is possible to use multiple parameters of the same kind, as well as different kinds of parameters:

```
automaton def X(event a, b; alg real c; event d):
  ...
end

event z;

x: X(z, z, 3 * 5, z);
```

Automaton definition `X` has four parameters: `a`, `b`, `c`, and `d`. Since `a` and `b` are both event parameters, a comma (`,`) is used to make sure the `event` keyword does not need to be repeated for parameter `b`. Algebraic parameter `d` is of a different kind, and is therefore separated using a semicolon (`;`).

Automaton instantiation `x` instantiates `X` with the event `z` (for the first, second, and fourth parameters), and value 15.0 (`3 * 5`, for the third parameter).

2.10.4. Groups

For large CIF specifications with many automata, it can be beneficial to add more structure to the specification. For this purpose, CIF has *groups*. Groups are named collections of automata and other declarations. For instance, consider the following CIF specification:

```
group factory:
  group hall1:
    automaton machine1:
      ...
    end

    automaton machine2:
      ...
    end
  end

  group hall2:
    automaton machine1:
      ...
    end

    automaton machine2:
      ...
```

```
    end
  end
end
```

This specifications features four automata that model the behavior of machines. The details of the actual automata are omitted, as they are irrelevant for this lesson. All four machines are in the same factory, but they are divided into two halls. This physical subdivision is expressed in the CIF specification using groups named `factory`, `hall1`, and `hall2`. While in this case the subdivision into a hierarchical specification structure is based on the physical subdivision of the actual system, the modeler is free to base the specification structure on other criteria.

Consider also the following partial CIF specification:

```
group configuration:
  const int  MAX_PRODUCTS        = 1500;
  const real MAX_PRODUCTION_TIME = 3.7;
  const real ARRIVAL_RATE        = 28.6;
end
```

The `configuration` group is used to group together several configuration values, modeled by constants. By grouping these constants together, it is more clear that they together are the configuration settings of the system, and that they belong together.

Finally, consider the following CIF specification, based on the non-linear system from the lesson on equations.

```
group tank:
  cont V = 5;
  alg real Qi;
  alg real Qo;

  equation V' = Qi - Qo;
  equation Qi = 1;
  equation Qo = sqrt(V);
end
```

By grouping the declarations and their equations together, it is becomes clear that together they model a `tank`. This is especially useful if other parts of the specification model something different.

2.10.5. Group definitions

Similar to how automaton definitions can be used for reuse of automata, group definitions can be used for reuse of groups:

```
automaton def Machine():
  location:
    initial;
```

```
  ...
end

group def Hall():
  machine1: Machine();
  machine2: Machine();
end

group def Factory():
  hall1: Hall();
  hall2: Hall();
end

factory1: Factory();
```

Automaton definition `Machine` models a machine, but most of the details are omitted here, as they are not relevant for this lesson. Group definition `Hall` models that each hall has two machines. Similarly, group definition `Factory` models that each factory has two halls.

Automata and groups are both *components* in CIF. Automaton definitions and group definitions can together be called *component definitions*. If we eliminate all component definitions and their instantiations, by replacing instantiations by their definitions, we get the following CIF specification:

```
group factory1:
  group hall1:
    automaton machine1:
      location:
        initial;
      ...
    end

    automaton machine2:
      location:
        initial;
      ...
    end
  end

  group hall2:
    automaton machine1:
      location:
        initial;
      ...
    end

    automaton machine2:
      location:
        initial;
      ...
```

```
    end
  end
end
```

Group definitions may be parametrized using the same kinds of parameters as automaton definitions.

Component parameters can be used inside component definitions to be supplied to other components. Consider for instance the following CIF specification:

```
automaton def Sensor():
  event go_on, go_off;

  location off:
    initial;
    edge go_on goto on;

  location on:
    edge go_off goto off;
end

sensor1: Sensor();
sensor2: Sensor();

automaton def Actuator(Sensor sensor):
  event turn_on, turn_off;

  location off:
    initial;
    edge turn_on  when sensor.on  goto on;

  location on:
    edge turn_off when sensor.off goto off;
end

group def Actuators(Sensor sensor):
  actuator1: Actuator(sensor);
  actuator2: Actuator(sensor);
  actuator3: Actuator(sensor);
end

actuators1: Actuators(sensor1);
actuators2: Actuators(sensor2);
```

Group definition `Actuators` models a collection of three actuators. The actuators behave as before. The actuators `actuator1`, `actuator2` and `actuator3` are provided parameter `sensor` as sensor. For the actuators part of `actuators1` the provided sensor is `sensor1` and for the actuators part of `actuators2` that is `sensor2`. Both `sensor1` and `sensor2` are actual sensors.

If we eliminate all component definitions and their instantiations, by replacing instantiations by their definitions, we get the following CIF specification:

```
automaton sensor1:
  ...
end
automaton sensor2:
  ...
end
group actuators1:
  automaton actuator1:
    event turn_on;
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor1.on goto on;
    location on:
      edge turn_off when sensor1.off goto off;
  end
  automaton actuator2:
    event turn_on;
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor1.on goto on;
    location on:
      edge turn_off when sensor1.off goto off;
  end
  automaton actuator3:
    event turn_on;
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor1.on goto on;
    location on:
      edge turn_off when sensor1.off goto off;
  end
end
group actuators2:
  automaton actuator1:
    event turn_on;
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor2.on goto on;
    location on:
      edge turn_off when sensor2.off goto off;
  end
  automaton actuator2:
    event turn_on;
```

```
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor2.on goto on;
    location on:
      edge turn_off when sensor2.off goto off;
  end
  automaton actuator3:
    event turn_on;
    event turn_off;
    location off:
      initial;
      edge turn_on when sensor2.on goto on;
    location on:
      edge turn_off when sensor2.off goto off;
  end
end
```

2.10.6. Imports

For large systems, having to model the entire system in a single CIF file can lead to long CIF specifications. Being able to split that single CIF file into multiple CIF files can help. Consider the following two CIF specifications in CIF files `producer.cif` and `consumer.cif` respectively:

```
// producer.cif

event provide;

automaton producer:
  location:
    initial;
    edge provide;
end
```

```
// consumer.cif

import "producer.cif";

automaton consumer:
  location:
    initial;
    edge provide;
end
```

The two CIF specifications together form a simplified producer/consumer system. The `producer.cif` file declares the `provide` event and the `producer` automaton. The `producer` can `provide` a product.

The `consumer.cif` file declares the `consumer` automaton, that can accept products provided by a producer. The `provide` event is not declared in that CIF specification. However, the `producer.cif` file is imported, which does declare that event. By importing another CIF specification, all declarations from that imported CIF specification (`producer.cif` in the example) become available in the importing CIF specification, i.e. in the specification that does the import (`consumer.cif` in the example).

The result of the import in `consumer.cif` is:

```
event provide;

automaton producer:
  location:
    initial;
    edge provide;
end

automaton consumer:
  location:
    initial;
    edge provide;
end
```

You can think of an import as being replaced by the content of the imported file. The `producer.cif` file contains only its own content, while the `consumer.cif` file contains the contents of both files, due to the use of the import.

If one CIF specification is merged into another CIF specification, the names of the declarations in both CIF specifications must be different. In general, it is not allowed to have declarations with the same name in multiple CIF specifications. For instance, in the example above, if the `provide` event were declared in both CIF specifications, the imports would be invalid. Exceptions to this rule are discussed in one of the next lessons, which explains the relation between imports and groups.

2.10.7. Imports and libraries

Using imports, it is possible to make libraries that can be used by multiple CIF specifications. For instance, consider the following CIF specification in file `math.cif`:

```
// math.cif

func int inc(int x):
  return x + 1;
end
```

This CIF specification declares a single function `inc` that takes an integer number and returns that number incremented by one. Now also consider the following CIF specification in file `counter.cif`:

```
// counter.cif

import "math.cif";

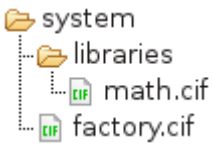
automaton counter:
  disc int count = 0;

  location:
    initial;
    edge tau do count := inc(count);
end
```

By importing the `math.cif` file, the `counter` automaton can use the `inc` function. Other CIF files could similarly import the `math.cif` file, essentially turning `math.cif` into a library.

It is possible to make a function library, consisting of commonly used functions, a constant library, with commonly used constants, or an automaton definition library, with automaton definitions. As libraries are just CIF files, they can contain anything as long as they are valid CIF files. Essentially, every CIF file that is imported in more than one other CIF file can be considered a library.

The import as used above, only works if both CIF files are in the same directory. Library files however, are often placed in a different directory. Consider the same two CIF files, but organized into directories (or folders) as follows:



Directory `system` contains a sub-directories named `libraries`, which contains the `math.cif` library. The `system` directory also contains the `counter.cif` file. The import in the `counter.cif` file needs to be adapted to refer to the `math.cif` file in the `libraries` directory:

```
// counter.cif

import "libraries/math.cif";

automaton counter:
  disc int count = 0;

  location:
    initial;
    edge tau do count := inc(count);
end
```

2.10.8. Imports and groups

Large systems can be hierarchically modeled using groups. When using imports, two specifications may not have declarations with the same name. Groups however, are the exception to this rule.

Consider a factory with two machines, each consisting of two parts. We can model this using five files, one for each of the parts of the machines, and one for the factory as a whole. The following CIF specifications show the contents of the five CIF files, where the comment at the first line indicates which file it is:

```
// p1.cif

group machine1:
  automaton part1:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end
```

```
// p2.cif

group machine1:
  automaton part2:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end
```

```
// p3.cif

group machine2:
  automaton part1:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end
```

```
// p4.cif
```

```
group machine2:
  automaton part2:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end
```

```
// factory.cif

import "p1.cif";
import "p2.cif";
import "p3.cif";
import "p4.cif";
```

The four CIF specifications for the machine parts differ only in their group and automaton names. Their implementations are kept identical for simplicity. The `factory.cif` file imports all four part specifications, which together form the full factory. The effect of the imports in `factory.cif` is the following:

```
group machine1:
  automaton part1:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end

  automaton part2:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end

group machine2:
  automaton part1:
    location idle:
      initial;
      edge tau goto running;
```

```
    location running:
      edge tau goto idle;
  end

  automaton part2:
    location idle:
      initial;
      edge tau goto running;

    location running:
      edge tau goto idle;
  end
end
```

For groups with the same name, the contents of the groups is merged together. That is, automaton `part1` in the `machine1` group from one CIF file, and the automaton `part2` in that same `machine1` group from another CIF file, end up in a single `machine1` group after eliminating the imports.

In general, contents of groups with the same name are merged into a single group. This works also for groups in groups, groups in groups in groups, etc. If two CIF files that are imported both contain a group `a` and in both CIF files those groups contain a group `b`, then the contents of both `a` groups are merged, and also the contents of both `b` groups are merged. It is not allowed for different CIF files to have declarations with the same name in the same group. It is allowed to have declarations with the same name in different groups. The file itself (the top level) can be considered a group as well.

2.10.9. Namespaces

When using imports and groups to model large hierarchical systems, it is a common pattern to have groups around the entire specification:

```
group factory:
  group machine1:
    group part1:
      automaton sensor1:
        ...
      end

      automaton actuator1:
        ...
      end
    end
  end
end
```

The groups around the entire specification lead to additional indentation. As this is a common pattern, the CIF language provides *namespaces* to reduce the impact of large numbers of top level

groups:

```
namespace factory.machine1.part1;

automaton sensor1:
  ...
end

automaton actuator1:
  ...
end
```

The `factory.machine1.part1` namespace has exactly the same effect as the original three groups. Both specifications can thus be considered identical.

2.10.10. Input variables

Input variables can be used to model that some data exists, without specifying the value or how or when the value changes. Input variables are used in cases where the CIF model is to be connected something else that provides the values.

Hardware example

The CIF model may for instance be a controller that is to be connected to a piece of physical hardware. The input variables then model sensors, for which the behavior is not modeled in the CIF model. As an example, consider the following CIF specification:

```
input bool sensor;

automaton controller:
  cont t der 1;
  alg bool actuator = on;

  location off:
    initial;
    edge when sensor do t := 0 goto waiting;

  location waiting:
    edge when not sensor goto off;
    edge when t > 1.5 goto on;

  location on:
    edge when not sensor goto off;
end
```

Input variable `sensor` models a digital sensor. When the sensor is on, the variable has value `true`

and when the sensor is off, the variable has value `false`. When the sensor is on and when it is off is not known, and thus not modeled in the CIF specification.

The `controller` waits (in location `off`) for the sensor to go on, and then resets continuous variable `t` to start measuring `1.5` time units (in location `waiting`). If that much time has passed, it goes to location `on`. If the sensor goes off then, or while it is waiting, it goes back to location `off`. Thus, the `controller` is only `on` if the sensor goes on, `1.5` time units have passed, and the sensor remains on.

The `actuator` algebraic variable models that an actuator is only on if the `controller` is in its on location. The `actuator` is then on if the sensor is one, and `1.5` time units have passed. The actuator is always turned off when the sensor is off.

Without a physical hardware system, the CIF specification is incomplete, as the value of the sensor is not known. The input variable acts as a placeholder for the actual sensor. If the CIF specification were to be coupled to a physical hardware system with a sensor and an actuator, the CIF model would then control the actuator based on the sensor values.

Co-simulation example

A CIF specification could model a part of a system, while another part of that same system is modeled in another system or language, such as Simulink. Consider for instance a tank filled with water. Water flows out of the tank constantly. By opening a value, water also flows in to the tank. Consider the following CIF specification:

```
input real level;

automaton valve:
  disc bool open = true;

  location:
    initial;
    edge when     open and level >= 10 do open := false;
    edge when not open and level <=  2 do open := true;
end
```

The water `level` is modeled as an input variable, meaning it is obtained from the Simulink model. The CIF specification models the valve, and the controller that opens and closes the value, based on the water level. It keeps the level between `2` and `10`.

The CIF model obtains the water level from Simulink by means of input variable `level`. It also provides the `open` variable of the `valve` automaton to Simulink. This variable can be used in Simulink to model how the water level changes over time.

Without the Simulink model, the CIF specification is incomplete, as the water level is not known. The input variable acts as a placeholder for the actual water level. If the CIF specification were to be coupled to a Simulink model that models the water level, the CIF specification would control the valve based on the water level.

CIF models example

Input variables can also be used to couple CIF models. Consider again the above water level controller example. It models the valve and controls the valve, but does not model the water level. The following CIF specification could be used to model the water level:

```
cont level der if valve.open: 1 else -1 end;

group valve:
  input bool open;
end
```

Continuous variable `level` models the water level. The water level increases (derivative `1`) if the valve is open, and decreases (derivative `-1`) otherwise. The `open` variable from the `valve` is used, but this variable is an input variable, and the value is not known in this specification. Input variable `open` acts as a placeholder for a value, and group `valve` acts as a placeholder for automaton `valve`.

The two CIF specifications both model a part of the system. One specification models the water level, and has a placeholder for the variable that indicates whether the valve is open. The other specification models the valve and controls it, but has a placeholder for the water level. If we couple the two CIF specifications, they provide each other with actual variables for each others input variables.

The two specifications can be coupled or *merged* together:

```
cont level = 0.0 der if valve.open: 1 else -1 end;

automaton valve:
  disc bool open = true;

  location:
    initial;
    edge when     open and level >= 10 do open := false;
    edge when not open and level <=  2 do open := true;
end
```

Using input variables to couple CIF specifications is somewhat similar to using imports to split a CIF specification over multiple files. There are however important differences.

The first difference is that when using imports, one CIF specification is coupled to another CIF specification. The coupling is hard-coded in the CIF specification, and can not be changed without changing the import. When using input variables and merging, it is possible to couple a CIF specification to a second CIF specification, and later couple it with a third, different specification instead. The decision about which models to merge can thus be postponed, and does not require changes to any of the CIF specifications.

Another difference is that with imports it is not allowed to have declarations with the same name in different CIF specifications (except for groups), while with the merger this is allowed in many situations. See the documentation of the CIF merger tool for further details on what can be merged using that tool.

SVG image interaction example

In later tutorial lessons, you will learn about SVG visualization and interaction. The CIF specification can be connected to an image. The image can then be adapted based on the state of the model, allowing for visualization. Additionally, you can set up clickable elements within the image that influence the model's behavior, allowing interaction with the modeled system through the image. The image then takes the role of the environment of the system being modeled.

An input variable is a connection to the environment, and the environment decides when and how the input variable changes value. If interaction via an image is used, clicks on certain elements of images can be specified to assign values to input variables. Specifically, SVG input mappings with updates allow for this.

As an example, consider an input variable that models a boolean sensor that indicates whether a user pushed a button:

```
input bool sensor;
```

Then the image could have a button, that when clicked assigns `true` to the input variable:

```
svgfile "system.svg";
svgin id "button" do sensor := true;
```

The details of this will become clearer during the later lessons on SVG visualization and interaction.

2.11. Stochastics

2.11.1. Stochastics

For many things in the world, the behavior is not the same every time. An example is a coin toss, where the result can be either heads or tails. Tossing a coin exhibits randomness. It is possible to model the different variants in a CIF model without specifying the likelihood of each of the outcomes. This approach was used in the lesson on non-determinism.

It is however also possible to explicitly specify the likelihood of each of the outcomes in the CIF model, using a *stochastic distribution* (also called probability distribution). A stochastic distribution describes how likely the different outcomes are. There are many different stochastic distributions. The Bernoulli distribution for instance, can be used to model things with two potential outcomes, such as a coin toss.

Consider the following CIF specification:

```
automaton coin_toss:
  disc dist bool d = bernoulli(0.5);
  disc bool outcome;

  location toss:
    initial;
    edge do (outcome, d) := sample d goto result;

  location result:
    edge when     outcome goto heads;
    edge when not outcome goto tails;

  location heads:
    edge tau goto toss;

  location tails:
    edge tau goto toss;
end
```

Variable `d` holds a stochastic distribution that produces boolean values (`true` or `false`), as indicated by its `dist bool` type. In this case it holds a Bernoulli distribution, with a probability of `0.5` (or chance of 50%) for `true`, and thus also the same probability/chance for `false` (the only other possible outcome).

The `bernoulli` function is used to create a Bernoulli distribution with the proper parameter (probability of `0.5` for `true`). Different distributions have different parameters. See the language reference documentation for further details. Distribution functions, such as the `bernoulli` function can be used to create distributions with specific parameters, and may only be used to initialize discrete variables.

Initially, the automaton is in its `toss` location, where the coin can be tossed. The edge uses the `sample` operator to get a *sample* from the bernoulli distribution. Each time the distribution is sampled, the `outcome` is either a `true` value or a `false` value. As we used a probability of `0.5` for both outcomes, half of the times `true` will be the outcome, and the other half of the times `false` will be the outcome, if we were to sample infinitely many times.

Sampling does not only result in the `outcome`, but also the distribution itself. This is further explained in one of the next lessons, which explains pseudo-randomness.

The result of sampling is stored in the `outcome` variable. In the `result` location, the sampling result is used to make a decision to go to either the `heads` location (`true` outcome) or the `tails` location (`false` outcome). From there, it is possible to go back to the `toss` location, to proceed with the next coin toss.

2.11.2. Discrete, continuous, and constant distributions

CIF features over a dozen different stochastic distributions. These distributions can be categorized

into three categories: discrete, continuous, and constant distributions. Discrete distributions can result in only a limited number of possible values. Examples include tossing a coin (heads and tails) and throwing a dice (six possible outcomes). Continuous distributions can result in all values from a certain range. Examples include the time it takes for a machine to produce a single product (positive amount of time), and the purity of produced medicine (0% to 100%). Constant distributions are discrete distributions that always result in the same sampled value. They are useful during the development of a model, or for debugging. The remainder of this lesson further explains these three categories, and illustrates them using some examples.

Discrete distributions

Discrete distributions can result in only a limited number of possible values. The lesson that introduced stochastics used the discrete Bernoulli distribution to model tossing of a coin, which can result in either heads or tails.

Throwing a dice can result in six possible outcomes: 1, 2, 3, 4, 5, and 6. For a fair dice, each of the outcomes is equally likely, with chance 1/6. This can be modeled using a discrete uniform distribution:

```
automaton dice:
  disc dist int d = uniform(1, 7);
  disc int result;
  disc list int results = [];

  location throw:
    initial;
    edge do (result, d) := sample d goto add;

  location add:
    edge do results := results + [result] goto throw;
end
```

The `uniform` function is used to create a uniform integer distribution that produces integer samples from the range [1..7) (inclusive on the left side, exclusive on the right side). Each sample is stored in `result` and added to the `results`. If we were to simulate this model forever, the `results` list would contain equal amounts of each of the six values.

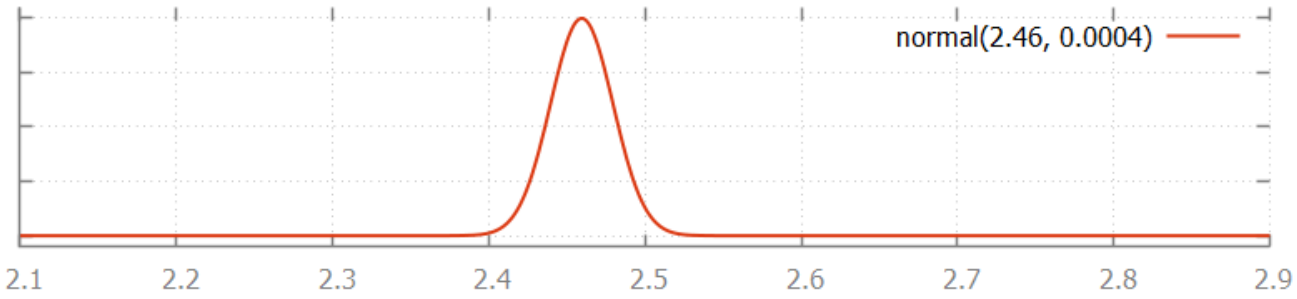
Continuous distributions

Continuous distributions return a value from a continuous range. An example is the weight of pennies. This could for instance be approximated using a continuous normal distribution with a mean of 2.46 grams and a variance of 0.0004 grams:

```
automaton pennies:
  disc dist int d = normal(2.46, 0.0004);
```

```
  location:
    initial;
    edge do (weight, d) := sample d;
end
```

The `normal` function is used to create a normal distribution with a mean of `2.46` and a variance of `0.0004`. The sampled weight is stored in the `weight` variable each time the distribution is sampled. The probability of the different weights as a result of the used normal distribution is depicted in the following figure:



Constant distributions

When developing a model with stochastic behavior, it can be hard to validate whether the model behaves correctly, since the stochastic results make it difficult to predict the outcome of experiments. As a result, errors in the model may not be noticed, as they hide in the noise of the stochastic results. One solution is to first make a model without stochastic behavior, validate that model, and then extend the model with stochastic sampling. Extending the model with stochastic behavior is however an invasive change that may introduce new errors. These errors can again be hard to find due to the difficulties to predict the outcome of an experiment. The constant distributions aim to narrow the gap by reducing the amount of changes that need to be done after validation.

With constant distributions, a stochastic model with sampling of distributions is developed, but the stochastic behavior is eliminated by temporarily using constant distributions. The model performs stochastic sampling of values, but with predictable outcome, and thus with predictable experimental results, making validation easier. After validating the model, the constant distributions are replaced with the distributions that fit the mean value and variation pattern of the modeled system, giving a model with stochastic behavior. Changing the used distributions is however much less invasive, making it less likely to introduce new errors at this stage in the development of the model.

Constant distributions always produce the same sampled value. Consider the following CIF specification:

```
automaton dice:
  disc dist int d = constant(3); // Constant distribution.
  disc int result;
```

```
  disc list int results = [];

  location throw:
    initial;
    edge do (result, d) := sample d goto add;

  location add:
    edge do results := results + [result] goto throw;
end
```

This model is identical to the dice model one used earlier in this lesson, except for distribution variable `d`, which now holds a constant distribution that only produces value `3` when sampled.

2.11.3. Pseudo-randomness

So far, the lessons on stochastics used the mathematical notion of stochastic distribution to describe how to model stochastic behavior. Simulating a model with stochastic behavior using a computer is however not stochastic at all. Computer systems are deterministic machines, and have no notion of varying results.

Pseudorandom number generator and seed

A pseudorandom number generator is used to create stochastic results instead. It starts with an initial seed, an integer number. From this seed, a function creates a stream of 'random' values. When looking at the values there does not seem to be any pattern. It is not truly random however. Using the same seed again gives exactly the same stream of numbers. This is the reason to call the function a pseudorandom number generator, as a true random number generator would never produce the exact same stream of numbers.

Simulation and initial seed

A sample of a distribution uses one or more numbers from the stream to compute its value. The value of the *initial seed* thus decides the value of all samples drawn in the simulation. By default, a different seed is used each time you run a simulation, leading to different results each time. You can also explicitly state what seed you want to use when simulating a model. See the Stochastic distributions and seeds page of the simulator for information on how to supply a custom seed for simulation.

While doing a stochastic simulation study, performing several experiments with the same initial seed invalidates the results, as it is equivalent to copying the outcome of a single experiment a number of times. On the other hand, when looking for the cause of a bug in the model, performing the exact same experiment is useful as outcomes of previous experiments should match exactly.

Sampling and tuple results

The seed determines the next sampled value. Whenever a distribution is sampled, the seed changes to the one that follows in the stream of random numbers generated by the pseudorandom number generator. This ensures that the next time the distribution is sampled, it results in a different sampled value.

In CIF, the seed is stored with the distribution and its parameters. Therefore, sampling does not only result in the sampled value, it also results in an updated distribution, with the same parameters, but a different seed value.

Consider the following CIF specification:

```
automaton coin_toss:
  disc dist bool d = bernoulli(0.5);
  disc bool outcome;

  location:
    initial;
    edge do (outcome, d) := sample d goto result;
end
```

The `bernoulli` function to create a Bernoulli distribution with parameter `0.5` (the chance for `true`). This Bernoulli distribution, along with its parameter and the seed, are stored in variable `d`. When the distribution in variable `d` is sampled (`sample d`), the result is both a sampled value, and a distribution (as a tuple). The distribution is the same distribution with the same parameters, but with an updated seed. The sampled value is usually stored in a variable (`outcome` in this case). The changed distribution is usually assigned to the same variable that contains the distribution that is sampled, to ensure that we can use the same variable for the next sampling. In other words, variable `d` is updated with the changed distribution each time it is sampled. The result of sampling is assigned to two variables, using tuple unpacking.

2.12. SVG visualization and interaction

2.12.1. SVG visualization

When simulating a model, it is often useful to be able to see the state of the model. For instance, the CIF simulator supports several forms of simulation output, such as printing state changes to the console. While this provides useful information, it is not easy to decipher the precise state of a (sub-)system. Also, it does not give a graphical overview of the current state of the system.

Therefore, CIF supports *SVG visualization*, which allows the user to make an image of the system, and have that image be updated during for instance simulation, based on the current state of the system. This allows the user to see changes in the state of the system, in a graphical representation of that system. This makes it much easier to get a global overview of the system during simulation. Since the user makes the image, the representation directly corresponds to the way the user sees the system.

Scalable Vector Graphics (SVG)

SVG visualization in CIF uses SVG images. SVG stands for Scalable Vector Graphics. SVG is a widely-deployed royalty-free standard for two-dimensional vector graphics, developed by the World Wide Web Consortium (W3C). CIF supports version 1.1 of the SVG standard.

SVG images consist of three types of objects: vector graphic shapes (rectangles, circles, etc.), raster

images, and text. The benefit of vector images formats over raster image formats, is that raster images are created with a fixed size, while vector images contain a description of the image and can be rendered at any size without loss of quality.

SVG image files are stored in an XML-based file format. This means that they can be edited with any text editor. However, it is often more convenient to edit them with a drawing program that supports vector graphics, such as Adobe Illustrator or Inkscape. Most modern web browsers can also be used to view SVG images.

Since SVG images are essentially XML files, we get all the benefits of XML for free. One of those benefits is that XML files represent tree structures, which can be easily manipulated by applications. This is essential for SVG visualization. Upon changes to the state of a CIF model, information from the model's state can be used to modify the XML tree. Changes to the XML tree then result in an updated SVG image. This way the image can be continuously adapted based on state changes, leading to an animated image.

In order to be able to modify particular objects (shapes, text labels, etc.) of the SVG image, the objects that make up the image should be identifiable. Here we once again reap the benefits of XML. All nodes (called elements) in the XML tree can have an `id` attribute that specifies the id (also called name or identity) of the element. As such, all SVG objects can be identified using their unique id.

The website of Jakob Jenkov has a very nice SVG Tutorial, its different elements and their attributes.

CIF/SVG declarations

The connection between the state of the CIF model and the SVG image can be specified by using CIF/SVG declarations (sometimes also called SVG declarations, for simplicity). Such declarations can be specified directly in the CIF model.

The following types of CIF/SVG declarations are available:

- An SVG file declaration specifies which SVG image file to use.
- A CIF/SVG output mapping specifies how to update a certain property of an element of an SVG image, based on the state of the model.
- A CIF/SVG copy declaration specifies that a part of an SVG image should be copied.
- A CIF/SVG move declaration specifies that a part of an SVG image should be moved.
- A CIF/SVG input mapping specifies how interaction with an SVG image can be used to affect the behavior of the model. SVG interaction is explained in a later tutorial lesson.

These declarations are all demonstrated in later lessons by means of examples. These examples also show some often used SVG elements and attributes, as well as some tips and tricks on how to handle common challenges.

Furthermore, examples of SVG visualization are available in other parts of the CIF documentation:

- Synthesis-based engineering in practice example

The details of the different CIF/SVG declarations can be found in the CIF reference manual.

2.12.2. First example

Let's look at a first example of SVG visualization. Here, we use a trivial example, which nonetheless gives a quick overview of what the idea of SVG visualization is all about, and what it looks like.

Note that it is not necessary to fully understand what exactly is going on, or how it works. Those details should become clear after reading the remaining pages of the documentation.

The SVG image

We use an SVG image with a single circle:



The element of the image that represents the circle is given id `lamp`, to be able to uniquely identify it. The SVG image is saved in a file named `lamp.svg`:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg width="70" height="70" xmlns="http://www.w3.org/2000/svg">
  <circle id="lamp" style="fill:black;" cx="35" cy="35" r="25"/>
</svg>
```

The SVG image has a size of 70 by 70 pixels if not scaled. It contains a single circle, with id `lamp`, `black fill` color, its center at x-coordinate `35` and y-coordinate `35`, and a radius of `25` pixels.

The CIF model

We use the following CIF model:

```
svgfile "lamp.svg";

automaton lamp:
  cont t der 1.0;

  location Off:
    initial;
    edge when t >= 1.0 do t := 0.0 goto On;

  location On:
    edge when t >= 2.0 do t := 0.0 goto Off;

  svgout id "lamp" attr "fill" value if Off: "gray" else "yellow" end;
end
```

This model describes not only the behavior of the lamp using a CIF automaton, but also contains

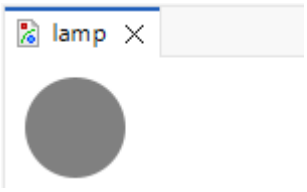
two CIF/SVG declarations.

The SVG file declaration at the start of the model connects the SVG image from the `lamp.svg` SVG file to the model. For this to work, the `lamp.svg` file should be in the same folder as the CIF model file.

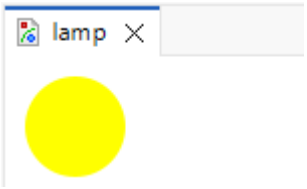
The CIF/SVG output mapping near the end of the model adapts a property of an element of the image, based on the state of the automaton. It sets the `fill` attribute of the element with id `lamp`, the circle of the image, to either of two colors: `silver` or `yellow`. If the automaton is in its `Off` location, the circle is silver. If the automaton is in its `On` location, the circle is yellow.

Simulation

If we simulate the CIF model, the automaton starts in its `Off` location, and thus we see a silver circle:



After a bit of time and taking a `tau` transition, the automaton is in its `On` location. We then get a yellow circle:



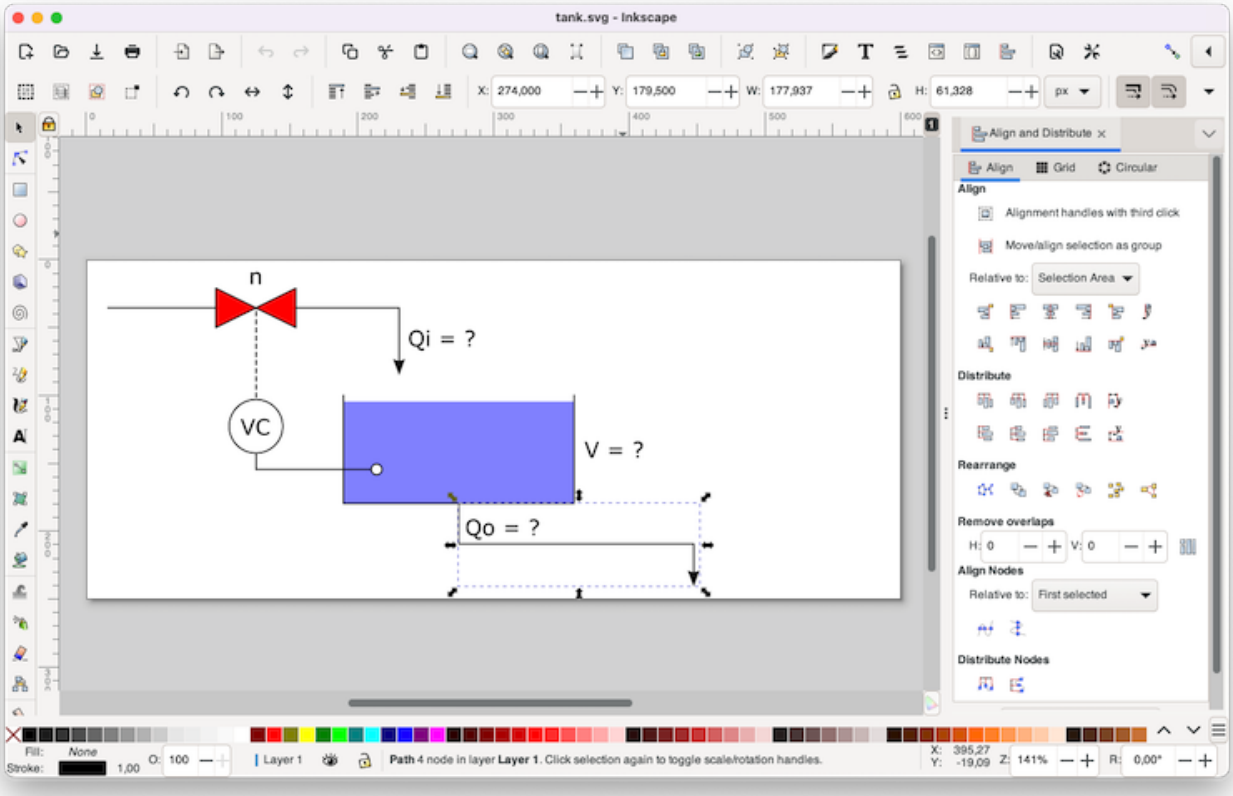
If we let the simulator run for a while, we see that the lamp is turned on and off repeatedly. SVG visualization essentially turns into an SVG movie of a flashing lamp.

2.12.3. Inkscape

SVG images can best be edited with a drawing program that supports vector graphics. One such program is Inkscape, a free and open source vector graphics editor that uses SVG as its native file format, and is available for many platforms, including Microsoft Windows, macOS, and Linux. In the CIF tutorial and the rest of the CIF documentation, we'll use Inkscape to create and edit SVG images.

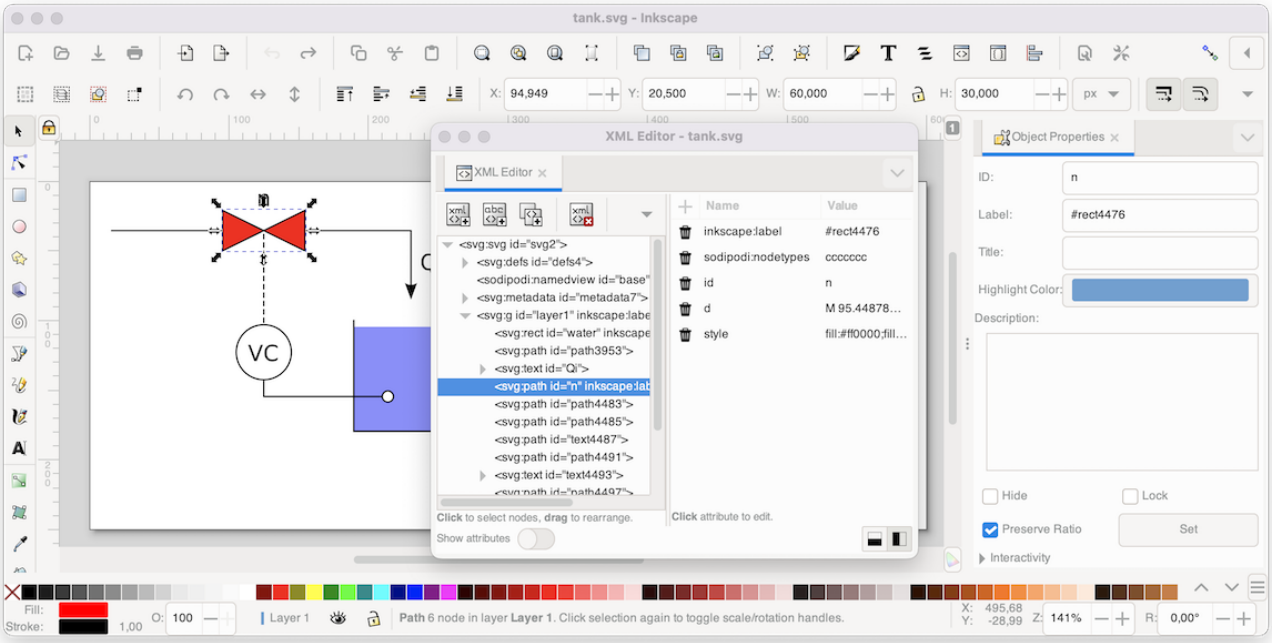
It is beyond the scope of this tutorial to explain how Inkscape works. For that, see the Inkscape website. Here, we focus on some of the specifics for working with Inkscape to create SVG images for SVG visualization of CIF models.

Here is a screenshot of the Inkscape interface:



Changing ids

SVG images consist of elements, also called or objects. Each object can be given an id, to be able to uniquely identify the object within the image. To change the id of an object in Inkscape, right click on the object, and choose **Object Properties...**. The *Object Properties* tab will appear on the right of the interface, which looks like this:



In this tab, you can change the object's id, by entering a new id in the *ID* field, and clicking the **[Set]** button.

XML editor

Inkscape also features an XML editor, that can be used to view and edit the underlying XML representation of the SVG image. The screenshot above also shows Inkscape's *XML Editor* as a window, which can be opened via **Edit › XML Editor...**. The XML editor can be useful to find out the exact structure of the XML tree, and to see the precise definitions that were used to represent the objects that you created on Inkscape's canvas. Panels like the *Object Properties* panel and the *XML Editor* panel can be fixed to the right side of the interface as tabs or be used as floating windows. Simply drag each panel to its preferred location.

Document size

It is recommended to always set the correct size of the image, before adding any shapes or text labels. To do this, select **File › Document Properties...**, to open the *Document Properties* window. On the *Display* tab, we recommend to set *Format* to *px* and *Display units* to *px*. You can then enter the *Width* and *Height* of the image. Finally, in the *XML Editor* panel, select the root SVG element (`<svg:svg`... node) by left clicking it, and delete the *viewBox* property by clicking the trash can icon in front of it. This ensures that the coordinates in the XML tree are the same as those displayed on the Inkscape interface.

Coordinate systems

The coordinate system used by SVG and Inkscape uses the upper left corner of the canvas as origin. Inkscape uses x and y coordinates of objects relative to the outside of their border (called *stroke* in Inkscape), while SVG uses x and y coordinates of objects relative to the middle of their border. It is good to be aware of this, when making CIF/SVG output mappings.

Text areas

Use regular text labels, rather than text areas, in your SVG images. When using the Inkscape text tool, select the tool, and left click once on the canvas to add a text label. After adding it, start typing text.

Inkscape also allows you to select the text tool, left click anywhere on the canvas, drag the mouse pointer to another position, and only then release the left mouse button. This creates a text area, where text is automatically wrapped to multiple lines, so that it stays within the bounds of the text area. CIF does not support text areas.

By default, newer versions of Inkscape create use the `shape-inside` CSS property for text areas, an

SVG version 2 feature not supported by CIF. With the *Use SVG2 auto-flowed text* option disabled, Inkscape uses SVG version 1 features for text areas, namely `flowRoot`, `flowRegion` and `flowPara` elements, but these are not supported by CIF either.

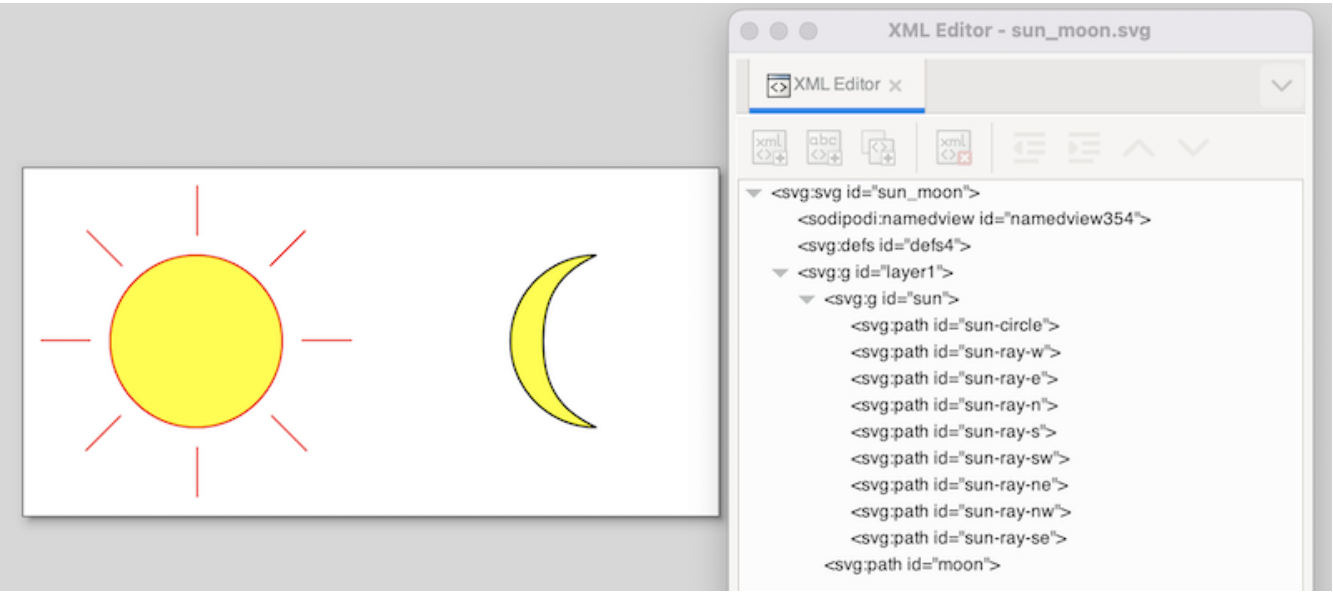
2.12.4. Sun/moon example

This SVG visualization example shows:

- Grouping of SVG objects.
- Controlling the visibility of objects based on the current locations of automata.

SVG image

The following SVG image is used for this example:



The image features a sun and a moon. The sun consists of a circle and several lines. They are grouped together, using the *Group* command from Inkscape's *Object* menu. The group can also be seen in the XML representation, as an `svg:g` element, with the actual shapes (paths in this case) as children. We assigned the `sun` id to the `svg:g` element, as it represents the entire sun. For clarity we gave the sub-elements ids as well, although that was not necessary, as we won't directly reference them in the CIF/SVG declarations.

CIF specification

The following CIF specification models the sun/moon example:

```
svgfile "sun_moon.svg";

automaton sun_moon:
  cont t der 1.0;

  location Day:
    initial;
    edge when t >= 12.0 do t := 0.0 goto Night;
```

```
  location Night:
    edge when t >= 12.0 do t := 0.0 goto Day;

  svgout id "sun" attr "visibility"
    value if Day: "visible" else "hidden" end;

  svgout id "moon" attr "visibility"
    value switch self:
      case Day:   "hidden"
      case Night: "visible"
    end;
end
```

Initially, it is day (location `Day`). It is day for 12 hours, after which it becomes night (location `Night`). After 12 hours, it becomes day again, etc.

There is one mapping to control the visibility of the sun, and one to control the visibility of the moon. Changing the visibility of the `svg:g` group element with id `sun` influences the visibility of all its children. This saves us from having to control the visibility of all the individual shapes that together represent the sun.

Both mappings map the current location of automaton `sun_moon` to either `"visible"` or `"hidden"`, although they do it in a different methods. The mapping for the sun uses an `if` expression, which is shorter in syntax, and especially suited for automata with only two locations. The mapping for the moon uses a `switch` expression over `self`, which is a way to refer to automaton `sun_moon`, from inside that automaton itself. The cases list all the possible locations, and map them to the corresponding visibility values. The `switch` method is a bit more explicit, and becomes more elegant as the automaton has more locations.

2.12.5. Walk example

The first part of this SVG visualization example shows a person walking on a floor, and features:

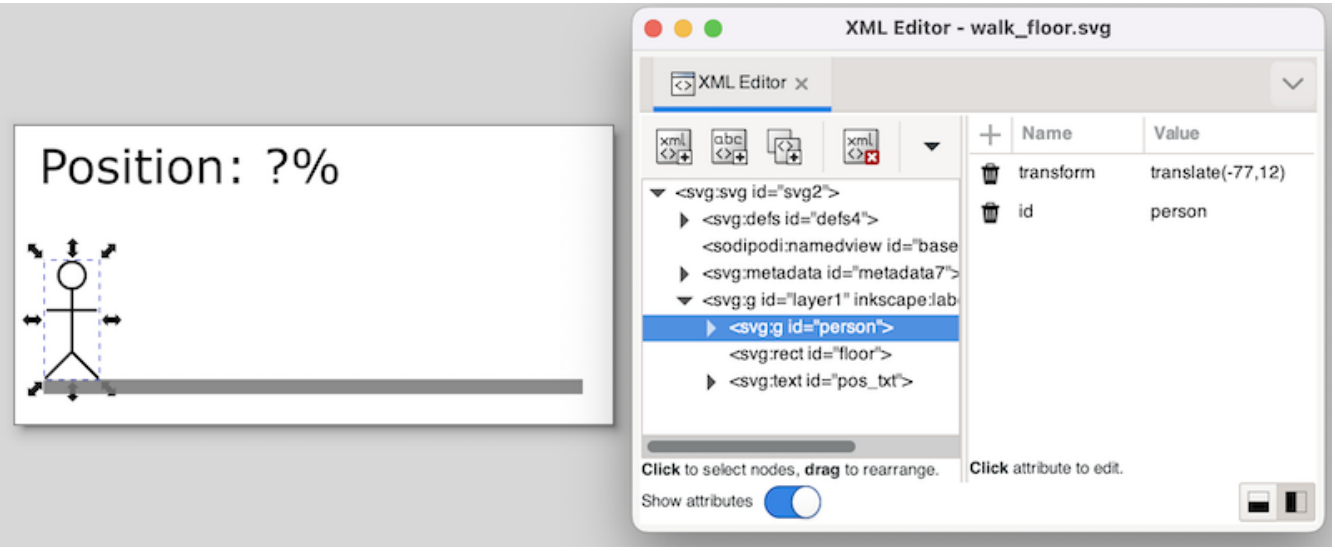
- Grouping of SVG objects.
- The position of grouped objects in SVG.
- Changing the position of grouped objects using an output mapping.
- Using the `scale` standard library function.
- Using text formatting.

The second part of this example shows a person walking around in a room, and features:

- Using multiple values in an output mapping.
- Using multiple format specifiers in a format pattern.

SVG image (walking on floor)

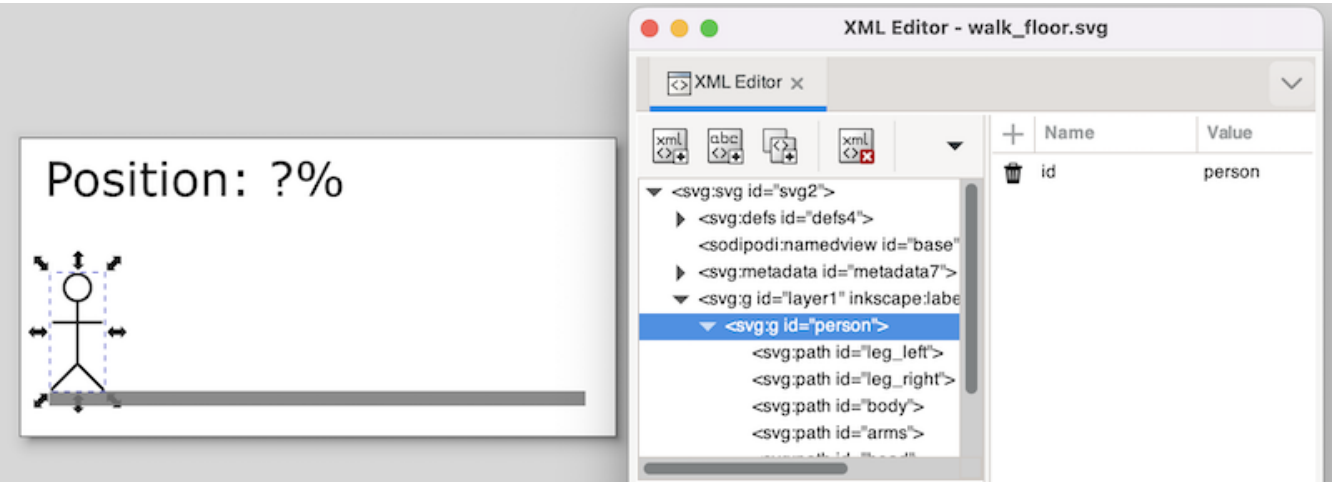
The following SVG image is used for the example of a person walking on a floor:



The image features a floor (gray bar), a person, and a text label. The person consists of some lines and a circle. These shapes are grouped together, using the *Group* command from Inkscape's *Object* menu. The group can also be seen in the XML representation, as an `svg:g` element, with the actual shapes (paths in this case) as children. We assigned the `person` id to the `svg:g` element, as it represents the entire person. For clarity we gave the sub-elements ids as well, although that was not necessary, as we won't directly reference them in the CIF/SVG declarations.

Note that we moved the person as a whole, to the start of the floor. Since `svg:g` elements don't have an `x` and `y` attribute in SVG, Inkscape applies a translation to move the group. This is reflected in the XML as a `transform` attribute, with the `translate(-77,12)` value. This indicates that the person was moved 77 pixels to the left, and 12 pixels to the bottom (as SVG has its origin in the upper left corner of the canvas).

To get rid of this, once the person is at the correct initial position, we can ungroup the `person` group (*Ungroup* command from Inkscape's *Object* menu), and regroup it. By ungrouping the group, the transformation is pushed to the individual elements of the group. The regroup simply adds a new group around the elements, without a translation transformation. The new group is given an automatically generated name, so we need to rename it to `person`. This ungroup/group trick is often very useful. After this trick, the image looks as follows:



The text label is to be used to show the position of the person, as a percentage. The left position is represented as 0%, and the right position is represented as 100%. The id of the text label is `pos_txt`.

CIF specification (walking on floor)

The following CIF specification models a person walking on a floor:

```
// Behavior.
automaton person:
  cont pos = 0.0;

  location forward:
    initial;
    equation pos' =  1.0;
    edge when pos >= 5.0 goto backward;

  location backward:
    equation pos' = -1.0;
    edge when pos <= 0.0 goto forward;
end

// Visualization.
svgfile "walk_floor.svg";

// width of the floor - width of the person = movement
// 180px              - 19px                = 161px
svgout id "person" attr "transform"
  value fmt("translate(%s,0)", scale(person.pos, 0, 5, 0, 161));

svgout id "pos_txt" text
  value fmt("%.1f%%", scale(person.pos, 0, 5, 0, 100));
```

A person starts at position (variable `person.pos`) zero (value `0.0`), and initially moves `forward`. After (s)he has reached position `5.0`, the movement is reversed. Once the person is back at the start position, the movement repeats itself.

There are two mappings, one that controls the horizontal position of the `person`, and another that controls the text of the text label.

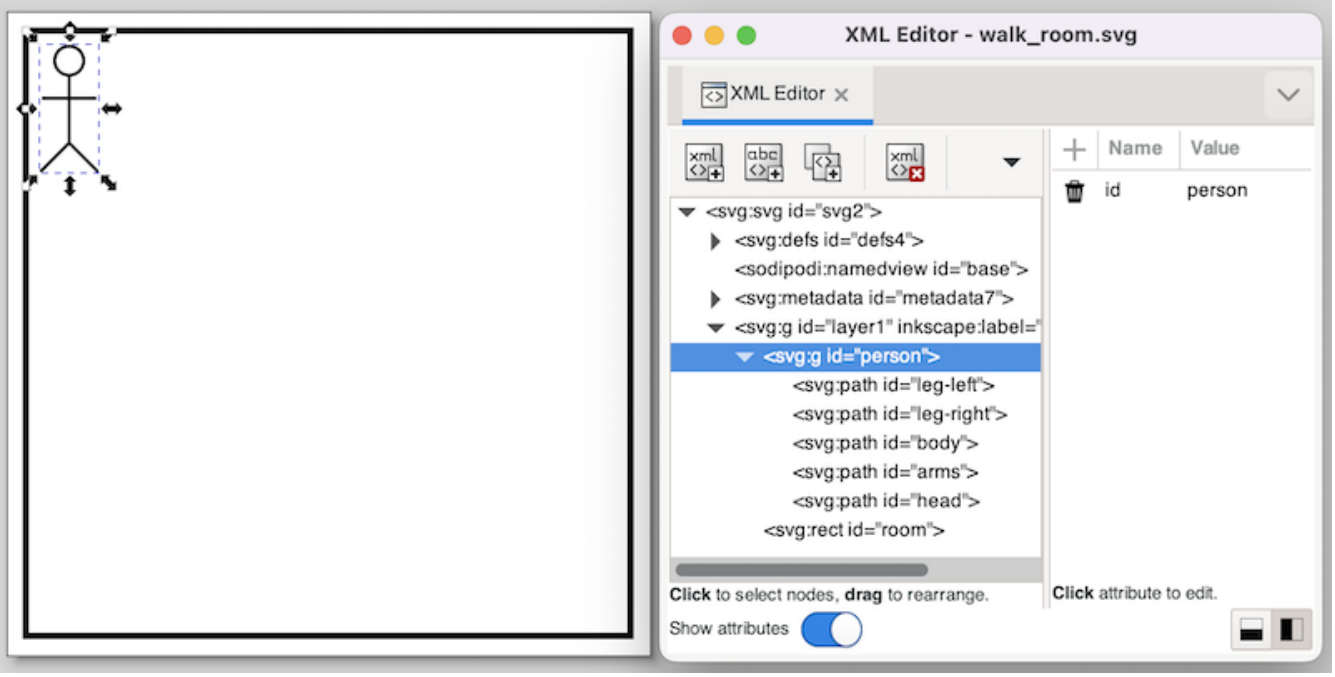
As stated above, the values of variable `person.pos` are in the range [0 .. 5]. As can be seen from the comments of the CIF model, the width of the floor and person are 180 pixels and 19 pixels respectively. The person can thus move 161 pixels to the right. The output interval of the `scale` standard library function is thus [0 .. 161]. As the vertical position is fixed, no vertical translation is required. The format pattern `"translate(%s,0)"` inserts the scaled value into the horizontal amount of the translation. The formatted output updates the `transform` attribute, making the person move horizontally.

The text of the text label with id `pos_txt` is updated using a second mapping. The value of variable `person.pos` is once again scaled using the `scale` standard library function. This time the output range is [0 .. 100], to get a percentage. Format pattern `"%.1f%%"` uses a `%.1f` format specifier to convert that percentage to a textual value, with a single digit after the decimal point. After the value of the percentage, a percentage character is included in escaped form (`%%`), to ensure it is not

interpreted as a format specifier.

SVG image (walking in room)

The following SVG image is used for the example of a person walking around in a room:



The image features a room (black rectangle) and a person. The person is the same group of elements as in the example of a person walking on a floor. The same group/ungroup trick is used here as well, to make sure the group has no translation transformation.

CIF specification (walking in room)

The following CIF specification models a person walking around in a room:

```
// Behavior.
automaton person:
  cont x = 0.0;
  cont y = 0.0;

  location right:
    initial;
    equation x' =  1.0;
    equation y' =  0.0;
    edge when x >= 5.0 goto down;

  location down:
    equation x' =  0.0;
    equation y' =  1.0;
    edge when y >= 5.0 goto left;

  location left:
    equation x' = -1.0;
    equation y' =  0.0;
```

```
    edge when x <= 0.0 goto up;

  location up:
    equation x' =  0.0;
    equation y' = -1.0;
    edge when y <= 0.0 goto right;
end

// Visualization.
svgfile "walk_room.svg";

// width of the room - width of the person = horizontal movement
// 180px             - 19px                = 161px
//
// height of the room - height of the person = vertical movement
// 180px              - 40x                  = 140px
svgout id "person" attr "transform"
  value fmt("translate(%s,%s)", scale(person.x, 0, 5, 0, 161),
                                scale(person.y, 0, 5, 0, 140));
```

A person starts at the top left position, and initially moves to the `right`. Once (s)he reaches the right side, (s)he start to move `down`, followed by a `left` and `up` movement. After the `up` movement, the person is once again at the top left position, start another walk around the room, etc.

There is only one mapping, which controls both the horizontal and vertical position of the `person`. The mapping is similar to the example of a person walking on a floor, except that now both the horizontal and vertical movements are included. The mapping uses two scaled values, one for the horizontal movement, and one for the vertical movement. The format pattern includes both values in the output. The first scaled value controls the horizontal movement, while the second scaled value controls the vertical movement.

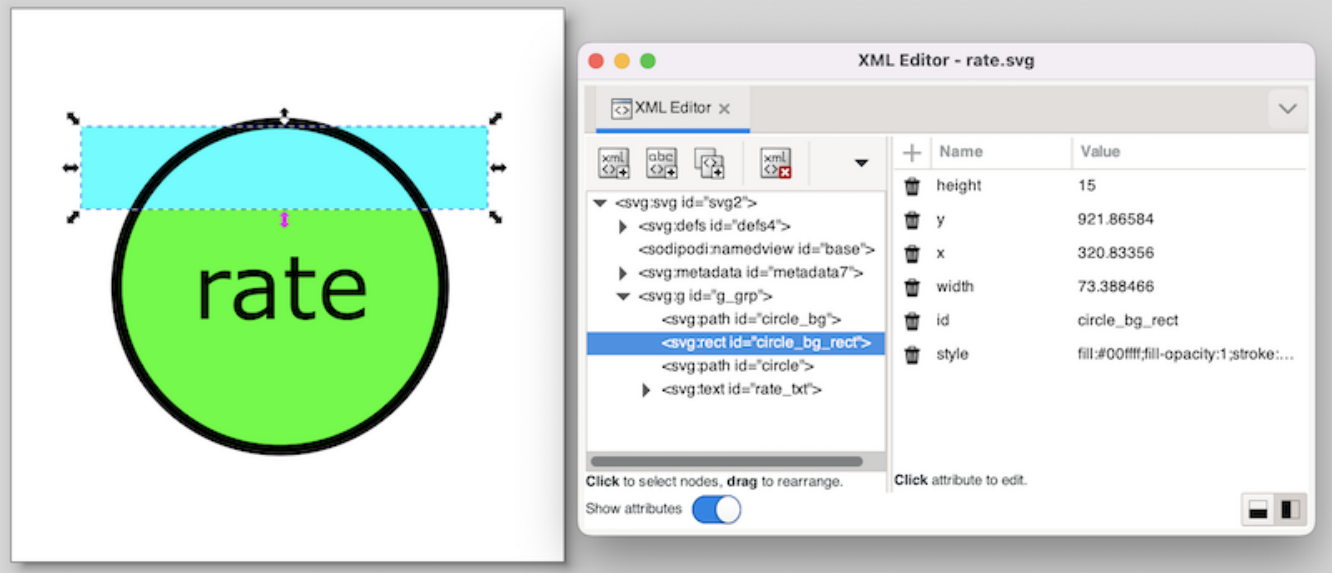
2.12.6. Rate example

This SVG visualization example shows:

- The use of literal values in output mappings.
- How it is best to use simple solutions to solve visualization problems.

SVG image

The following SVG image is used for this example:



The SVG image visualizes a machine as a circle. The rate of the machine is visualized as the background of the circle. The higher the rate, the more the circle is filled.

Instead of working with complicated clippings or linear gradients, we use a more simple solution here. The black circle has a transparent background (`fill` attribute set to `none`). This is the SVG element with id `circle`. Behind it is the same circle (id `circle_bg`), but without a border (`stroke` attribute set to `none`), and with a green fill color. Between those two circles, is a rectangle (id `circle_bg_rect`), that at the top is aligned to the top of the circle. By changing the height of the circle, we can now control what part of the green background circle is obscured by the rectangle.

During the actual visualization, we'll use a white fill color for the rectangle. Since the background is also white, this 'hides' the rectangle, and makes for a partial green background of the circle. Because a white rectangle can not be distinguished from the background, we used a cyan fill color for the rectangle. This makes that we can easily edit the rectangle, and see whether it has the correct initial position, size, etc.

CIF specification

The following CIF specification models the rate example:

```
// Behavior.
const real MARGIN = 0.01;

automaton machine:
  cont rate der if rate < target_rate:  1.0
                else                   -1.0
                end;
  disc real target_rate = 0.5;
  disc dist real rate_sampler = uniform(0.0, 1.0);

  location:
    initial;
    edge when rate - MARGIN <= target_rate and target_rate <= rate + MARGIN
         do (target_rate, rate_sampler) := sample rate_sampler;
end
```

```
// Visualization.
svgfile "rate.svg";

svgout id "circle_bg_rect" attr "height"
  value scale(machine.rate, 0, 1, 58, 0);

svgout id "circle_bg_rect" attr "fill" value "white";
```

This rather artificial machine has a `rate` at which it produces. The rate is a value in the interval [0 .. 1]. A target rate (variable `target_rate`) is randomly chosen in the same interval. The rate is automatically adapted to get closer to the target rate. As soon as the rate is close enough to the target rate, a new target rate is randomly selected, using the `rate_sampler`, which is a continuous uniform stochastic distribution.

The first mapping controls the `height` of the rectangle. The value of variable `machine.rate` has a value in the interval [0 .. 1]. This interval is mapped to the [58 .. 0] pixels interval. That is, if the rate is 0, the height of the rectangle is 58 pixels, and it completely covers the background of the circle. The circle then appears with a completely white background. If the rate is 1, the height of the rectangle is 0, and a completely green background is shown. For rates in between 0 and 1, the lower part of the background of the circle is green, and the upper part is white.

The second mapping sets the `fill` color of the rectangle to `white`.

The following image shows an example of the visualization of a partial rate:



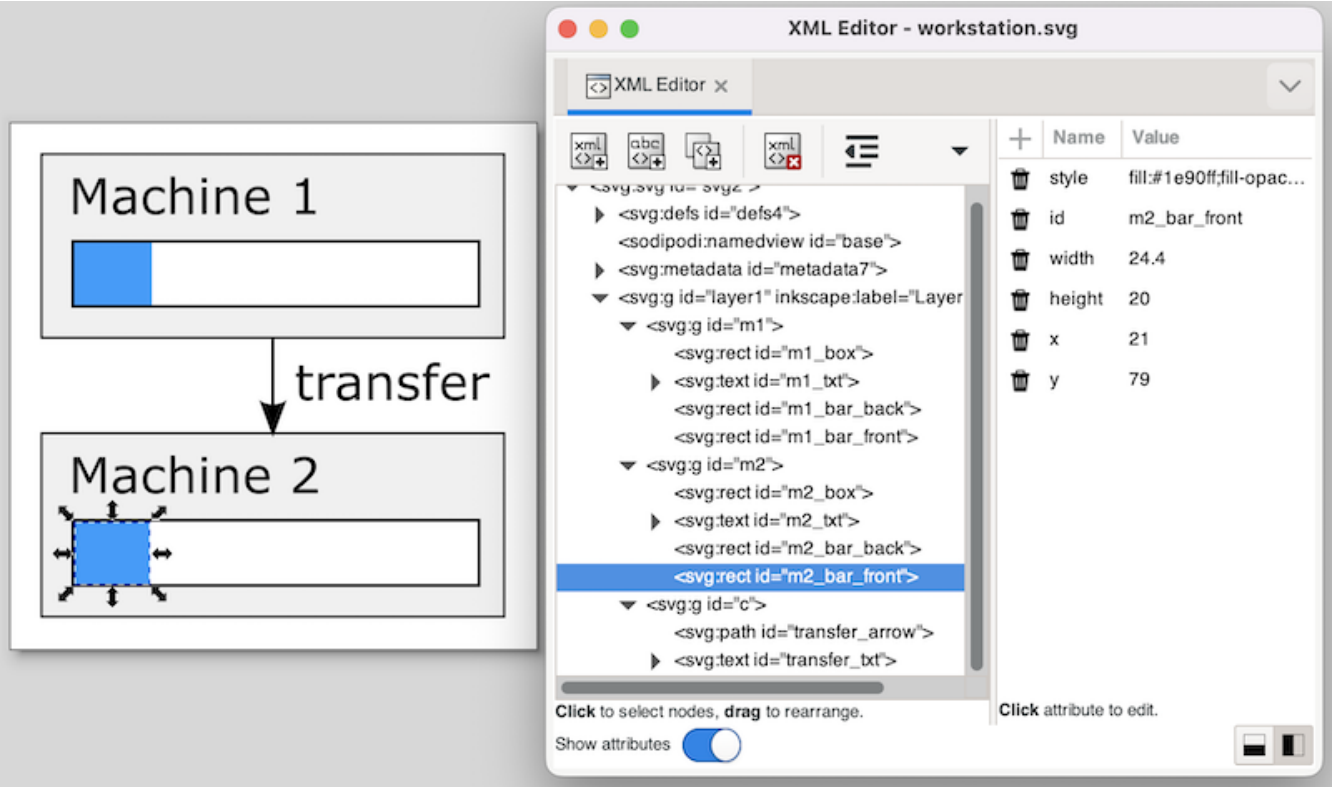
2.12.7. Workstation example

This SVG visualization example shows:

- Basing the width of an object on the value of a clock in one location, and setting it to a fixed value in another location.
- Changing the fill color of a rectangle based on the current location of an automaton.
- Updating multiple attributes of an SVG element, using multiple output mappings.
- Specifying colors in SVG.
- Improving scalability by making use of the reuse of mappings.

SVG image

The following SVG image is used for this example:



In the image, two machines are clearly visible. Each machine has a bar that shows the progress of the machine. Looking at the *XML Editor* window, we can see the bar has a background (box), and a foreground (front) rectangle. The `width` and `fill` color of the latter will be updated to reflect the machine's status.

CIF specification

The following CIF specification models the workstation example:

```
svgfile "workstation.svg";

event generate, transfer, exit;

m1: Machine(1, 1.0, generate, transfer);
m2: Machine(2, 2.5, transfer, exit);

automaton def Machine(alg int nr; alg real ptime; event start, done):
  cont t der 1.0;

  location idle:
    initial;
    edge start do t := 0.0 goto processing;

  location processing:
    edge when t >= ptime goto completed;

  location completed:
    edge done goto idle;
```

```
  svgout id "m" + <string>nr + "_bar_front" attr "width"
    value scale(if processing: t else ptime end, 0, ptime, 0, 130);

  svgout id "m" + <string>nr + "_bar_front" attr "fill"
    value if idle : "gray" else "dodgerblue" end;
end
```

The example models a workstation with two machines. The first machine has a processing time of `1.0` time units and the second machine has a processing time of `2.5` time units. Each machine gets products to `start` processing, processes them, and sends them out as it is `done`. The first machine gets its input from an infinite generator (event `generate`). The second machine sends its finished products to an infinite exit buffer (event `exit`). Products are transported from the first machine to the second machine via a transport system (event `transfer`). Note that the `Machine` automaton definition has three locations. In location `idle` it is waiting for a product to enter the machine (event `start`). In location `processing` it processes the product, which takes `ptime` time units. Finally, in location `completed` it has completed processing the product, and waits until it can send it as output. After that, the machine is back in the `idle` location, where it waits for the next product.

Both machines (`m1` and `m2`) are instantiations of the same automaton definition (`Machine`). The definition/instantiation mechanism allows us to specify the behavior of a machine only once, and still have multiple machines. This reuse improves scalability, as well as maintainability. The reuse of the behavior applies to the visualization as well. By declaring the output mappings in the automaton definition, we get mappings for each instantiation of an actual machine.

Since each machine has its own graphical representation in the SVG image, the graphical elements for the different machines have different ids as well. However, they only differ in the number that is used. For instance, there is an `m1_bar_front` rectangle for the first machine, and an `m2_bar_front` rectangle for the second machine. In order to make sure each machine gets the correct mappings that apply to their own ids, we added an algebraic parameter `nr` to the definition of `Machine`, that indicates to the machine which machine it is. We can then use that `nr` to compute the id. For machine `m1`, the `Machine` automaton definition is instantiated with value `1` for the `nr` parameter. The `"m" + <string>nr + "_bar_front"` id expression then evaluates to `"m1_bar_front"`, and `m1_bar_front` is used as the id for the output mapping.

The first mapping updates the width of the `m*_bar_front` rectangles to show the progress of the machines. If a machine is actually `processing`, the value of clock `t` is used, as that clock indicates how much of the processing has been completed.

When a machine is `idle` (and waiting for a product) or has `completed` processing (but can not yet push the processed product out of the machine), the progress is either nothing, or everything. In both cases, we choose the full processing time (variable `ptime`) as the value. The reason to choose this value for the `idle` location, becomes clear in the next output mapping.

The `ptime` value is chosen for the `completed` location to ensure that the rectangle does not get wider than the box (rectangle `m*_bar_back`). If we would have used the value of clock `t` in this location as well, then since the value of the clock keeps increasing in that location, the value becomes larger than `ptime`, and thus the progress bar would be partly outside the box (rectangle `m*_bar_back`).

The value that is selected for this first mapping, is transformed using the `scale` standard library

function. The function is used to scale the processing time interval [0 .. `ptime`] to the [0 .. 130] interval, which represents the `width` of the rectangle in pixels. The upper bound of 130 pixels matches the `width` of the box (rectangle `m*_bar_back`).

The second mapping updates the `fill` color (foreground color) of the same SVG element. In the `idle` location, the color is `gray`. Since we used the full width in that location (value `ptime` in the [0 .. `ptime`] interval), the entire box is then filled by a gray rectangle, indicating that the machine is idle.

For the other locations, the `fill` color is set to `dodgerblue`. Since the width in pixels is a value in the interval [0 .. 30], part of the box will be blue, and part will be the background color of the box, which is white. In the `completed` location, the full width is used as well, resulting in a completely blue box, indicating that the machine has a completely processed product, waiting to be released as output.

Colors in SVG

In the mappings, as described above, we used names to identify colors. SVG however, provides several ways to specify colors:

- Named colors, such as `red`, `yellow`, `lightblue` or `burlywood`.
- Decimal RGB (red-green-blue) colors, such as `rgb(255,0,0)` for `red`, or `rgb(255,255,0)` for `yellow`.
- Hexadecimal RGB (red-green-blue) colors, such as `#f00` or `#ff0000` for `red`, or `#ff0` or `#ffff00` for `yellow`.
- RGB (red-green-blue) colors in percentages, such as `rgb(100%,0%,0%)` for `red`, or `rgb(100%,100%,0%)` for `yellow`.

Every color can be specified by one of the three RGB color variants. The colors that have a predefined color name, can be specified in four different ways.

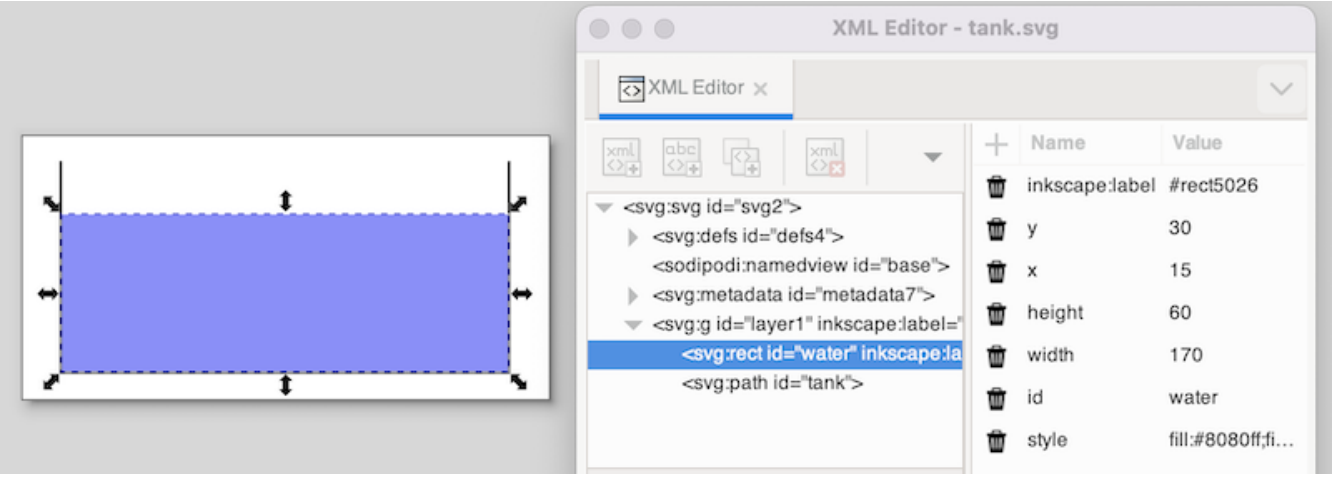
2.12.8. Tank example

This SVG visualization example shows:

- Use of the `scale` standard library function.
- How inverting the y-axis can reduce the number of output mappings.

SVG image

The following SVG image is used for this example:



The image features a tank (black lines) and the water in the tank (blue rectangle). The water rectangle is selected. In the *XML Editor* window, you see the position as SVG represents it. It shows `15` for the `x` position and `30` for the `y` position. These values are relative to the upper left corner of the canvas.

CIF specification

The following CIF specification models the tank example:

```
// Behavior.
const real CAPACITY   = 10.0;
const real MAX_LEVEL  =  8.0;
const real INIT_LEVEL =  5.0;
const real MIN_LEVEL  =  2.0;

automaton tank:
  cont level = INIT_LEVEL;

  location open:
    initial;
    equation level' =  1.0;
    edge when level >= MAX_LEVEL goto closed;

  location closed:
    equation level' = -1.0;
    edge when level <= MIN_LEVEL goto open;
end

// Visualization.
svgfile "tank.svg";

svgout id "water" attr "height" value scale(tank.level, 0, CAPACITY, 0, 80);
```

In this specification, we have a tank filled with water. Initially, the `level` (height) of the water is `5.0`, and the valve is `open`. The level linearly increases with a rate of `1.0` per time unit. Once the level is `MAX_LEVEL`, we close the valve, and go to the `closed` location. Here the level linearly decreases until we reach the minimum level (`MIN_LEVEL`), after which we open the valve again. The level controller

keeps the level between the minimum and maximum allowed levels.

There is only one mapping, and it controls the height of the `water` rectangle. The water level in the model stays between zero and `CAPACITY`, by definition (as a higher level would overflow). This results in an input interval of [0 .. `CAPACITY`]. The height of the tank in the SVG image is `80` pixels. The output interval thus becomes [0 .. 80].

Simulation

When we simulate the tank example using the CIF specification and SVG image as specified above, we immediately observe a problem:

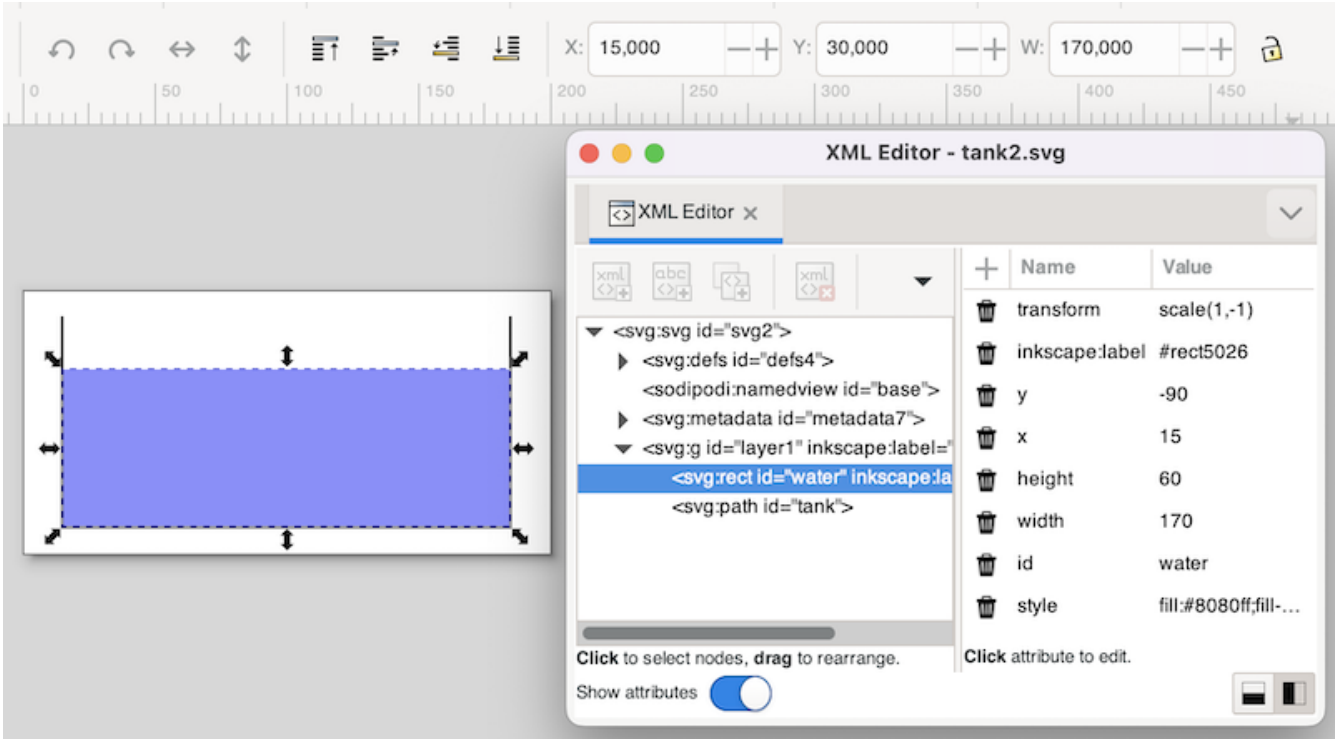


In SVG the origin is in the upper left corner of the canvas. Changing the height of an object results in the top of the object being fixed, and the bottom of the object being closer to or farther away from that top. What we want in this case, is to have the bottom of the object fixed, and the top of the object being closer to or farther away from the top, depending on the height of the object.

SVG image (revisited)

A naive solution would be to add a second output mapping, to correct the vertical position (`y` attribute) of the `water` rectangle. This is a valid solution, and it works. There is however a simpler and more elegant solution.

Consider the following, updated SVG image:



This SVG image is the same as the original SVG image for this example, but with an inverted `y`-axis,

for the `water` rectangle. The `y`-axis was inverted by using the *Flip Vertical* command from Inkscape's *Object* menu, while the `water` rectangle was selected. As can be seen in the screenshot, Inkscape corrects for the applied transformation, and still indicates the same `x` and `y` coordinates (at the top of the screenshot). In SVG (and thus the *XML Editor* window), we see a new `transform` attribute, `scale(1,-1)` as value, which represents the inversion of the `y`-axis. Also, the bottom of the water (the inverted top of the water) is now `90` pixels (the inverted `-90` value of the `y` attribute) from the top of the image. Using this vertical flip, we now fixated the 'bottom' of the water, and we can thus change the height of the water without having to correct its position. The mappings for this modified SVG image are the same as the original mappings.

2.12.9. Lamps example

This SVG visualization example shows:

- Scalability of CIF models and SVG visualizations.
- The use of CIF/SVG copy declarations.
- The use of CIF/SVG move declarations.

Introducing the lamps example

Consider the following CIF specification:

```
// Behavior.
automaton def Lamp():
  event toggle;
  disc bool on = false;

  location:
    initial;
    edge toggle do on := not on;
end

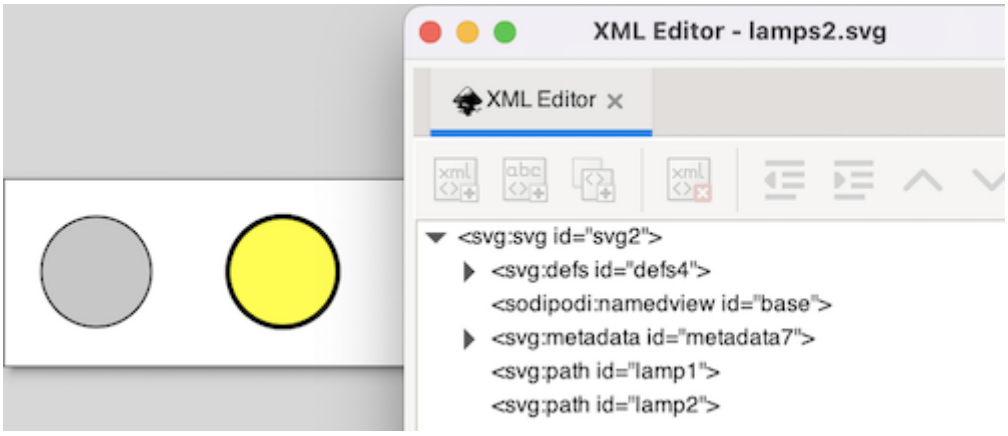
lamp1: Lamp();
lamp2: Lamp();

// Visualization.
svgfile "lamps2.svg";

svgout id "lamp1" attr "fill" value if lamp1.on: "yellow" else "silver" end;
svgout id "lamp2" attr "fill" value if lamp2.on: "yellow" else "silver" end;

svgout id "lamp1" attr "stroke-width" value if lamp1.on: 5 else 2 end;
svgout id "lamp2" attr "stroke-width" value if lamp2.on: 5 else 2 end;
```

This specification has two lamps, `lamp1` and `lamp2`. Both lamps have identical behavior, and therefore the behavior is only specified once, using automaton definition `Lamp`. This definition is instantiated two times, once for each of the lamps. A lamp is initially off, can go on, go off again, go on again, etc. The following SVG image is used for this example:



The visualization shows the two lamps. The first lamp is off, and has a silver background color. The second lamp is on and has a yellow background color. The lamp that is off has a thin border and the lamp that is on has a thicker border. The CIF specification also contains four CIF/SVG output mappings, two for the background colors of the circles and two for the thickness of the borders (strokes) of the circles.

SVG visualization scalability issues

The use of the automaton definition allows us to add another lamp by simply adding the following line to the CIF specification:

```
lamp3: Lamp();
```

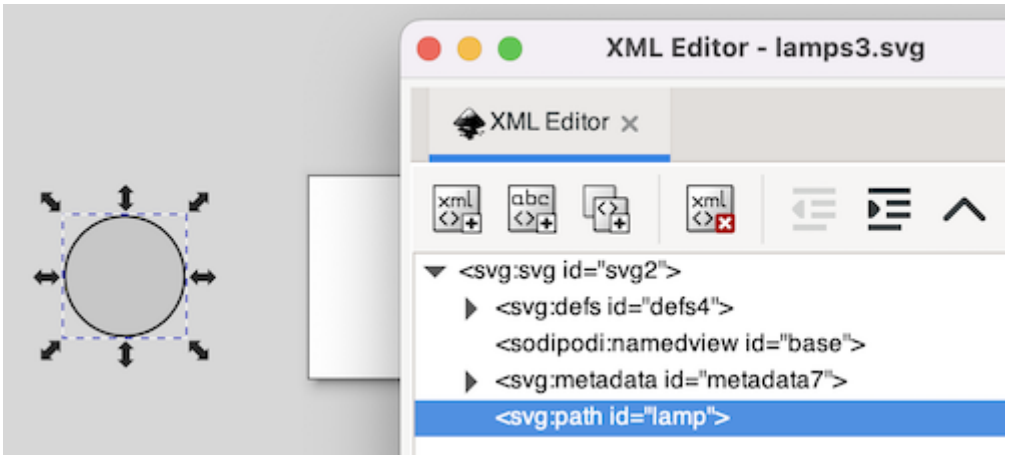
This single line ensures that we have a third lamp, with the corresponding behavior. However, we also need to adapt the SVG image, by adding another circle, and we need to add two more output mappings to the CIF model. In this case, the additional amount of work is manageable. If however our lamp would have been represented in the image by many different graphical elements, adding another lamp would require copying all those graphical elements, and might also lead to adding many more output mappings per lamp.

Furthermore, if we were to decide to change the graphical representation of lamps from a circle to a square, we would have to change the graphical representation of each of the lamps in the image.

It should be clear that this approach does not scale very well. Ideally, we would add another lamp by just adding another instantiation of the `Lamp` definition as we did above, and automatically get another circle in the image, and two more output mappings in the CIF model.

A scalable solution

For the behavior of the lamp, we already had a scalable solution, by using the component definition/instantiation mechanism. That same mechanism can be used to make the image and mappings scalable. First, we'll change the SVG image:



The second lamp has been removed from the image. We only keep one lamp, as a sort of template for all the lamps. Since we want to use the same graphical representation for all lamps, we only want to include it once in the SVG image. This allows us to change the graphical representation of the template, to have all the lamps change appearance at once. The lamp has been moved outside of the visible part of the canvas, and its `id` has been changed from `lamp1` to `lamp`. The reason for these last two changes will become clear once we look at the new CIF model, and especially the new output mappings:

```
svgfile "lamps3.svg";

automaton def Lamp(alg int nr):
  // Behavior.
  event toggle;
  disc bool on = false;

  location:
    initial;
    edge toggle do on := not on;

  // Visualization.
  svgcopy id "lamp" post <string>nr;

  svgmove id "lamp" + <string>nr to 20 + (nr - 1) * 100, 20;

  svgout id "lamp" + <string>nr attr "fill"
    value if on: "yellow" else "silver" end;

  svgout id "lamp" + <string>nr attr "stroke-width"
    value if on: 5 else 2 end;
end

lamp1: Lamp(1);
lamp2: Lamp(2);
lamp3: Lamp(3);
```

The first change is that the `Lamp` automaton definition now has a `nr` parameter, that indicates which lamp it is. Each lamp is given a different number when instantiated. This allows the lamp automata

to know their own identity, and use that for the SVG visualization.

We also put the CIF/SVG declarations for the visualization inside the automaton definition. By doing this, we ensure that each instantiation (each lamp) gets their own version of these declarations.

The CIF/SVG declarations have changed as well. A CIF/SVG copy declaration has been added. The copy declaration states that the SVG element with id `lamp` (the circle) should be copied. It also states that the copy should be postfixed with `<string>nr`. That is, for automaton `lamp1`, the value of the `nr` parameter is `1`, and the id is thus postfixed with `"1"`. In other words, circle `lamp` is copied, and the copy is given name `lamp1` (composed from `lamp` and `1`). For `lamp2`, the id of the copied circle is `lamp2`, etc. For each of the lamps, the copy gets a different name, that is unique within the SVG image. This single copy declaration takes care of the scalability of the SVG image, by ensuring we have exactly as many circles as we have lamps (not counting the template circle `lamp`).

The copies of the template circle all get exactly the same attributes as the template (except for their `id` attribute). As such, they are all positioned outside the visible part of the canvas. The easiest way to put them at the correct positions, is to use a CIF/SVG move declaration, which is also new compared to the previous CIF model. The move declarations use the ids of the copied circles, so that each move declaration moves the circle for that specific lamp. They are all moved to 20 pixels from the top of the canvas, and '20 + (`nr` - 1) * 100' pixels from the left side of the canvas. So, the first lamp (`nr` 1) is moved to (20, 20), the second lamp (`nr` 2) is moved to (120, 20), and the third lamp (`nr` 3) is moved to (220, 20). This single move declaration takes care of the scalability of the positions of the lamps, by ensuring we have exactly as many movements as we have lamps. By using the `nr` parameter, each lamp can easily be moved to their own unique position.

The output mappings for the background color of the circles and thickness of the borders of the circles, are very similar to the output mappings of the first version of the CIF model, though there are a few changes. The mappings use the same method as the move declaration, to construct the ids. Since the mappings are now defined within the automaton definition, they can directly refer to the `on` variable. This ensures that for each different lamp, the mappings refer to the `on` variable of that specific lamp. We now have only one output mapping for the background color, and also only one output mapping for the thickness of the border.

By putting the CIF/SVG declarations inside the automaton definition, and by using the copy and move declarations, we now have a scalable solution. We only have one lamp in our SVG image, only one behavioral specification in the CIF model, and only one output mapping per attribute that we want to couple. No matter how many instantiations of the `Lamp` automaton definition we add, we don't have to manually change the SVG image or the visualization coupling.

For completeness, take a look at this CIF model after the elimination of component definition/instantiation, the elimination of algebraic variables, and the simplification of values.

```
automaton lamp1:
  event toggle;
  disc bool on = false;
  svgcopy id "lamp" post "1";
  svgmove id "lamp1" to 20, 20;
  svgout id "lamp1" attr "fill" value if on: "yellow" else "silver" end;
  svgout id "lamp1" attr "stroke-width" value if on: 5 else 2 end;
```

```
  location:
    initial;
    edge toggle do on := not on;
end
automaton lamp2:
  event toggle;
  disc bool on = false;
  svgcopy id "lamp" post "2";
  svgmove id "lamp2" to 120, 20;
  svgout id "lamp2" attr "fill" value if on: "yellow" else "silver" end;
  svgout id "lamp2" attr "stroke-width" value if on: 5 else 2 end;
  location:
    initial;
    edge toggle do on := not on;
end
automaton lamp3:
  event toggle;
  disc bool on = false;
  svgcopy id "lamp" post "3";
  svgmove id "lamp3" to 220, 20;
  svgout id "lamp3" attr "fill" value if on: "yellow" else "silver" end;
  svgout id "lamp3" attr "stroke-width" value if on: 5 else 2 end;
  location:
    initial;
    edge toggle do on := not on;
end
svgfile "lamps3.svg";
```

2.12.10. Buffers/products example

This SVG visualization example shows:

- Improving hierarchical scalability by making use of the reuse of mappings.
- Improving hierarchical scalability by making copies of copied elements.

It is assumed the reader is already familiar with the workstation example.

SVG image

The following SVG image is used for this example:



The SVG image contains a single template for a buffer (large rectangle), with one product (small rectangle). The template is moved outside of the canvas to ensure the template itself is not shown.

The SVG file has the following XML content:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="220" height="70" id="root" version="1.1">
  <g id="buffer">
    <rect width="100" height="50" x="-120" y="10" id="background"
style="fill:#ff0000"/>
    <rect width="10"  height="30" x="-110" y="20" id="product"
style="fill:#8b0000;visibility:hidden;"/>
  </g>
</svg>
```

We'll use CIF to turn this into two buffers with three products each, in a scalable manner. That is, to add a single buffer, we only need one more instantiation, and also to put an additional product in each buffer, we also only need one more instantiation.

CIF specification

The following CIF specification models the buffers/products example:

```
svgfile "buffers_products.svg";

buffer0: Buffer(0);
buffer1: Buffer(1);

group def Buffer(alg int nr):
  svgcopy id "buffer" post <string>nr;

  svgmove id "buffer" + <string>nr to 10 + nr * 110, 10;

  product0: Product(nr, 0);
  product1: Product(nr, 1);
  product2: Product(nr, 2);
end

group def Product(alg int bufferNr, productNr):
  svgcopy id "product" + <string>bufferNr post "_" + <string>productNr;

  svgout id fmt("product%d_%d", bufferNr, productNr)
         attr "transform"
         value fmt("translate(%d,0)", productNr * 20);

  svgout id fmt("product%d_%d", bufferNr, productNr)
         attr "visibility"
         value "visible";
end
```

Two buffers are present (`buffer0` and `buffer1`). Both are instances of the `Buffer` definition, and are

provided with their own identity (parameter `nr`). For each buffer, the `buffer` element of the image is copied, and the copy is given a unique number. For instance, for instantiation `buffer0`, `nr` is `0`, and thus `buffer` is copied to `buffer0`, `background` is copied to `background0` and `product` is copied to `product0`.

Each buffer is also moved. The first buffer (`nr 0`) gets moved to position (10, 10), while the second (`nr 1`) gets moved to position (120, 10).

Each buffer contains three products, as the `Product` definition is instantiated three times. The number of the buffer is passed along, as is a unique product number. For the copy of the products, the copy of the product in the copied buffer is copied, and given a unique name using the product number. For `buffer0`, the copy of the `product` was already named `product0`. This already copied element is copied another three times, to `product0_0` (by the copy declaration in CIF group `buffer0.product0`), `product0_1` (by the copy declaration in CIF group `buffer0.product1`), and `product0_2` (by the copy declaration in CIF group `buffer0.product2`). Similarly, three copies are made for `buffer1`.

Unlike the buffers, which are moved to an absolute position using an `svgmove`, the products are moved relatively, using an `svgout` for their `transform` attribute. The first product (`product0_0`) is not moved, the second product (`product0_1`) is moved `20` pixels, etc.

The buffer template was put outside of the canvas to show only the copies and not the template. For the products, which are contained in the buffers, this trick is not possible. So, instead the `visibility` of the template `product` was set to `hidden` in the original SVG image. To show the actual (copied) products, an `svgout` is used to set the `visibility` attribute to `visible`. The template `product` is not changed, so it remains hidden.

The result of all this hierarchical 2-level structure is the following SVG image:



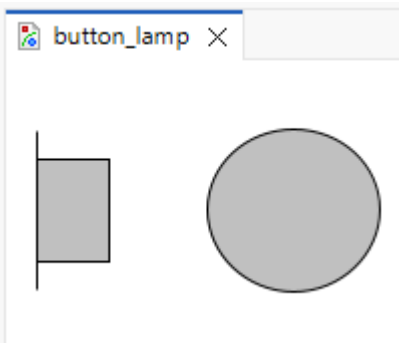
2.12.11. SVG interaction

SVG interaction allows SVG images that are used for SVG visualization to be used to interactively control the model. That is, SVG interaction allows certain elements of SVG images to be made interactive, such that clicking them has an effect on the model. If we consider that SVG visualization allows to have an SVG image be updated based on the state of the model, mapping the model state to the image, then SVG interaction kind of works the other way around, mapping clicks of elements in the image to effects on the behavior of the model.

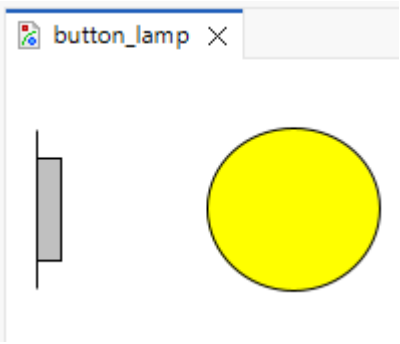
The idea of SVG interaction is that certain SVG elements are coupled (mapped) to events or updates in the CIF specification. Clicking on those graphical elements then results in the corresponding event being taken or update being executed. This way, the user interface of a system can be modeled graphically, and can be coupled to the specification, by means of events and updates. The user interface of the system is thus part of the specification, but is controlled interactively (at runtime) by the user.

Example

As an example, consider a button and a lamp. Clicking the button turns on the lamp. Clicking the button again off the lamp. We can use SVG visualization and interaction to model this, which allows for interactive simulation. Initially, the button (on the left) is unpushed (the wide rectangle) and the lamp (on the right) is off (silver color):



Clicking the button pushes it (it becomes less wide) and turns on the lamp (yellow color):



CIF/SVG input mappings

Interactivity is specified by means of CIF/SVG input mappings. There are two types of input mappings: those that map to events and those that map to updates.

An input mapping may map to an event, allowing a click on an interactive element to map to executing a single transition for an event. The events that are coupled to visualization elements are called *SVG interactive events* or *SVG input events*.

An input mapping may also map to updates, allowing a click on an interactive element to lead to input variables being updated by assigning them (new) values.

Single event input mapping

The simplest form of an SVG input mapping maps to a single event. For instance:

```
svgin id "button" event machine.button.toggle;
```

This input mapping maps the SVG element with id `button` to the `machine.button.toggle` event from the CIF specification. Specifying an input mapping for an SVG element turns that SVG element into

an *interactive SVG element*. In this case, the `machine.button.toggle` event is disabled until the element with id `button` is clicked. Each click enables the event, allowing at most one transition for it. If the event is enabled in the rest of the model, a transition is taken for it, and the event becomes disabled again. If the event is not enabled in the rest of the model, for instance because some automaton with the event in its alphabet can't synchronize in its current location, then the click is ignored and the event becomes disabled again. The event remains disabled until the `button` element is clicked again.

`if` event input mapping

Sometimes, the simple *single event input mapping* is not enough. Consider for instance the following CIF specification:

```
automaton button:
  event u_pushed, u_released;

  location Released:
    initial;
    edge u_pushed goto Pushed;

  location Pushed:
    edge u_released goto Released;
end
```

This specification models a `button` that is initially released (location `Released`). When the button is pushed (event `u_pushed`), the location is updated to `Pushed`. If we now want to couple a graphical representation of a button to this CIF specification, the event to choose depends on the current location of the `button` automaton. This can be mapped using an *if event input mapping*, as follows:

```
svgin id "button" event
  if   button.Released: button.u_pushed
  elif button.Pushed:   button.u_released
  end;
```

If the button is released, the `u_pushed` event is chosen, and if the button is pushed, the `u_released` event is chosen.

The `if` event input mappings are essentially an `if` expression that result in an event rather than a value. The conditions (such as `button.Released`) are guard expressions that evaluate to a boolean value. The guard expressions may be arbitrary expressions, similar to the use of expressions in output mappings.

If a condition holds, the corresponding event is chosen. The entries are processed in the order they are specified: if the first guard (of the `if`) holds, the first event is chosen, otherwise if the second guard (of an `elif`) holds, the second event is chosen, etc. The `elif` parts are optional, and you may include as many as you want. The last part of the `if` is optionally allowed to be an `else`, to indicate

that if none of the previous guards hold, the event of the `else` is to be chosen.

Update input mapping

An SVG input mapping can also update the values of input variables. For instance:

```
input int counter;

svgin id "increase" do counter := counter + 1;
svgin id "decrease" do counter := counter - 1;
```

In this example, by clicking the SVG element with id `increase` the value of `counter` is increased by one. Similarly, clicking the SVG element with id `decrease` decreases the value of `counter` by one.

Be aware that the initial value of an input variable cannot be set in the specification. Instead, CIF tools like the CIF simulator allow to set the initial value via an option.

2.13. Text

2.13.1. Print output

At runtime, when simulating or executing a model, it is often useful to be able to see the state of the model. For instance, the CIF simulator supports several forms of simulation output, such as printing state changes to the console. While this provides useful information, it can also be useful to generate custom output, tailored to one's specific needs. To allow this, the CIF language supports *print output*.

Conceptually, simulations and executions result in traces. A trace consists of the states that are encountered and the transitions that are taken, during the simulation or execution. The print output declarations allow specifying for which transitions text should be printed (for instance only for transitions for a certain event), when the text should be printed (for instance before or after the transition), where the text should be printed (for instance to a file), and what text should be printed.

Print output declarations are an extension to the CIF language, and can be specified directly in the CIF specification. They are only used to couple the model to textual output. They don't influence the behavior of the model itself.

Print output can be used to generate text that after simulation or execution is further processed, for instance for visualization or for analysis using statistical software. Print output can also be useful for debugging.

Quick example

This example serves only to give a quick taste of what print output declarations look like:

```
printfile "out.txt";

print x for e;
```

This example prints the value of variable `x` to a file `out.txt` each time after a transition for event `e` is taken.

In the next lesson, you'll learn more about print declarations and how to use them, by means of examples.

2.13.2. Print output examples

This lesson explains print output by means of examples. It does not explain all uses of print output, and all details related to print output. For those details, see the reference manual's print declarations documentation instead.

In this lesson, only the output from print declarations is shown when console output is presented. All other output, such as that from the CIF simulator, is omitted.

Before we start with the examples, there is a section that introduces the example model and trace on which the examples are based. Then, the following examples are explained in detail:

- Example 1: Printing the buffer count for all states
- Example 2: Printing to a file
- Example 3: Printing the odd/even status for all states
- Example 4: Printing only for odd states
- Example 5: Printing the result of adding an item
- Example 6: Printing addition/removal results
- Example 7: Printing value changes
- Example 8: Printing headers and footers
- Example 9: Printing the new time after time passes
- Example 10: Printing all unique time values
- Example 11: Printing transitions to/from a location

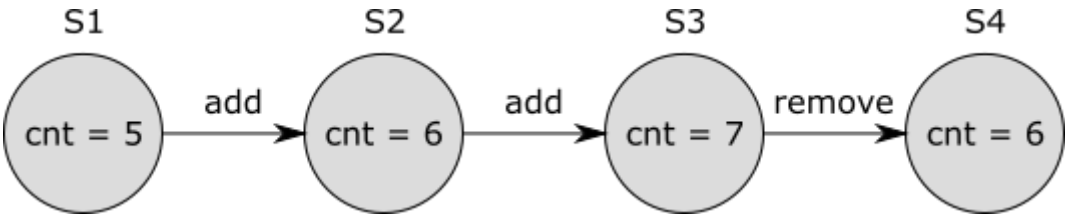
Example model and trace

The examples of print output in this lesson all make use of the same CIF model:

```
automaton buffer:
  event add, remove;
  disc int cnt = 5;
```

```
  location:
    initial;
    edge add    when cnt < 10 do cnt := cnt + 1;
    edge remove when cnt >  0 do cnt := cnt - 1;
end
```

The `buffer` initially contains five items (`cnt = 5`). As long as there is room in the buffer (`cnt < 10`), an item can be added (event `add`) to the buffer. As long as the buffer is not empty (`cnt > 0`), an item can be removed (event `remove`) from the buffer. The model allows to choose between adding an item to the buffer, and removing an item from the buffer. The following is a potential simulation/execution trace:



In the initial state (S1), variable `cnt` has value `5`. An item is then added (transition for the event `add`), resulting in a new state (S2), where `cnt` has value `6`. After that, another item is added, leading to state S3, where `cnt` has value `7`. Finally, an item is removed by means of a transition for event `remove`, leading to state S4, where `cnt` has value `6` again.

Example 1: Printing the buffer count for all states

If we want to print for every state (S1 through S4), the number of items in the buffer, we could add the following print declaration to automaton `buffer`:

```
print cnt;
```

Then, as a result of this print declaration, the following text is printed to the console:

```
5
6
7
6
```

That is, for every state (S1 through S4), the value of variable `cnt` is printed, on a new line.

Example 2: Printing to a file

Consider the following print declarations:

```
print cnt     file "a.txt";
```

```
print cnt * 2 file "b.txt";
```

The first print declaration results in the number of items in the buffer being printed for every state of the trace, to a file named `a.txt`. The second print declaration prints the number of items in the buffer multiplied by two, to a file named `b.txt`. These declarations use local file declarations to specify the file to which output is to be written. It is also possible to use global print file declarations:

```
printfile "ab.txt";
print cnt;
print cnt * 2;
```

The print file declaration declares that output of this scope is to be printed to a file named `ab.txt`. The two print declarations don't specify a local file declaration, and thus use the default file for their scope, in this case file `ab.txt`. Alternating lines with the number of items in the buffer and twice that amount are thus printed to that file.

Example 3: Printing the odd/even status for all states

If we want to print for every state, whether the buffer contains an odd or even number of items, we could add the following print declaration to automaton `buffer`:

```
print if cnt mod 2 = 0: "even" else "odd" end;
```

The print declaration uses an `if` expression. Then, as a result of this print declaration, the following text is printed to the console:

```
odd
even
odd
even
```

Example 4: Printing only for odd states

If we want to print the number of items in the buffer, but only for states where the number of items is odd, we could add the following print declaration to automaton `buffer`:

```
print cnt when cnt mod 2 = 1;
```

Then, as a result of this print declaration, the following text is printed to the console:

```
5
7
```

Example 5: Printing the result of adding an item

If we want to print the result of adding an item, that is the new buffer count after a transition for the `add` event, we could add the following print declaration to automaton `buffer`:

```
print cnt for add;
```

Then, as a result of this print declaration, the following text is printed to the console:

```
6
7
```

Since twice an item is added to the buffer, two lines of text are printed.

Example 6: Printing addition/removal results

If we want to print the resulting buffer count after every change to the buffer (addition or removal), we could add the following print declaration to automaton `buffer`:

```
print cnt for add, remove;
```

Then, as a result of this print declaration, the following text is printed to the console:

```
6
7
6
```

Two items are added to the buffer, and one item is removed from the buffer, leading to three lines of text being printed.

Since the `add` and `remove` events are the only events in the system, and no edges exist without an event (which would implicitly use the `tau` event), the print declaration can also be specified as follows:

```
print cnt for event;
```

That is, the value of variable `cnt` is printed after each event transition.

Example 7: Printing value changes

If we want to print the buffer count before and after removal of an item from the buffer, we could add the following print declaration to automaton `buffer`:

```
print pre cnt post cnt for remove;
```

As a result of this print declaration, the following text is printed to the console:

```
7
6
```

As only one item is removed for our example trace, two lines of text are printed, one with the buffer count before the removal (due to `pre cnt`), and one with the buffer count after the removal (due to `post cnt`).

Example 8: Printing headers and footers

If we want to print for every state (S1 through S4), the number of items in the buffer, we could add the following print declaration to automaton `buffer`:

```
print cnt;
```

If we want to print some text before this, as a header, and some text after this, as a footer, we could add the following print declaration to the top level scope of the specification, or to automaton `buffer`:

```
print "header" for initial;
print "footer" for final;
```

As a result of the original print declaration, and these two new print declarations, the following text is printed to the console:

```
header
5
6
7
6
footer
```

Example 9: Printing the new time after time passes

In the example trace given at the top of this page, no passage of time is included. If however a trace includes time passage, it is possible to filter printing to only time transitions. For instance, if we want to print the new time after time passes, we could use the following print declaration:

```
print time for time;
```

This prints the value of variable `time`, every time after a time transition. The first `time` in the print

declaration denotes that the value of variable `time` should be printed, and the second `time` denotes that it should be printed only after passage of time (after time transitions). As an example, the following could be printed to the console:

```
0.005
0.75
3.1
7.9
```

Example 10: Printing all unique time values

In the ninth example, we printed the new time value after each time transition. However, initially time is zero. Since the initial state is not reached via a time transition, time zero is not printed. We could adapt the print declaration to the following, to print all unique values of variable `time`, for all states of the trace:

```
print time for initial, time;
```

This initially prints the value of variable `time`, and also prints it after every time transition. As an example, the following could be printed to the console:

```
0.0
0.005
0.75
3.1
7.9
```

Example 11: Printing transitions to/from a location

In the example model given at the top of this page, automaton `buffer` has only one location. If however an automaton has multiple locations, it may be useful to print the effect of transitions ending in a certain location, or the state of the system for transitions starting in a certain location. Consider the following print declaration:

```
print x for event when aut.loc;
```

This prints the value of variable `x` after event transitions leading to a state where location `loc` is the current location of automaton `aut`. That is, whenever an event transition leads to entering that location, the value of variable `x` after that transition is printed. Since self loops have the same source and target location, this includes self loops. To exclude self loops, use the following print declaration:

```
print x for event when pre not aut.loc post aut.loc;
```

This print declaration prints the same text, but only prints it after event transitions from a state where location `loc` is not the current location of automaton `aut` to a state where location `loc` is the current location of automaton `aut`.

So far in this example, we printed text for transitions ending in a location. To print text for transitions starting in a location, consider the following print declaration:

```
print pre y for event when pre aut.loc post not aut.loc;
```

This print declaration prints the value of variable `y` in the state that is exited by the transition, for all event transitions that start in location `loc` of automaton `aut`, but do end in that same location.

2.13.3. Text formatting

This lesson explains text formatting by means of examples, in the context of print output. It does not explain all uses of text formatting, nor all details of text formatting. For those details, see the reference manual's text formatting details documentation instead.

In this lesson, only the output from print declarations is shown when console output is presented. All other output, such as that from the CIF simulator, is omitted.

While we explain text formatting in the context of print declarations, text formatting by means of the `fmt` function can also be used to create string values in other contexts.

Introduction

Consider the following CIF specification:

```
print "Time=" + <string>time;
```

When simulated, it could give the following output:

```
Time=0.0
Time=3.333333333333336
Time=5.000000000000001
Time=10.0
```

As you can see, due to rounding and simulation imprecision, the length of the textual representations of the values of `time` can vary wildly during simulation. Furthermore, the explicit casts (conversions) from real to string (`<string>`) and string concatenation (the `+` operator) can quickly become cluttered, and hard to read.

To improve readability of the print declaration, we could adapt the CIF specification to the following:

```
print fmt("Time=%s", time);
```

Here, manual text construction has been replaced by the use of the `fmt` standard library function. The first argument is a *format pattern*, and the remaining arguments are the values that are to be included in the text, in this case the value of variable `time`. The `fmt` function ensures that we no longer have to use casts. We also no longer have to create pieces of text and concatenate them (using the `+` operator).

This is a typical use of a format pattern. The print declaration prints the value that results from text formatting. The value that is used is the value of variable `time`. The value of this variable is not used 'as is', but is instead converted to a textual representation using the format pattern. This format pattern specifies that the resulting text should start with `Time=`. It also specifies that the value (of variable `time`) should be included at the end of the textual representation. The `%s` part of the format pattern is a *format specifier*. The `%s` specifier is the generic specifier that can convert any type of value to a textual representation.

When simulated, this new specification gives the exact same output as the previous version.

Real value formatting

As we saw in the previous section, due to rounding and simulation imprecision, the length of the generic textual representations of the values of variable `time` can vary wildly during simulation. This reduces readability of the results, as a lot of irrelevant digits are included. Consider the following alternative CIF specification:

```
print fmt("Time=%.2f", time);
```

Here, instead of a `%s` specifier, a `%f` specifier is used. This specifier can only be used for `real` numbers, and indicates that the number should be formatted as a floating point number in decimal representation. The `.2` part specifies the precision, and indicates that the floating point number in decimal notation should have exactly two digits after the decimal point. When simulated, this could give the following output:

```
Time=0.00
Time=3.33
Time=5.00
Time=10.00
```

Putting values in columns

In the examples above, we included a single value in the output. Now consider the following CIF specification:

```
print fmt("%.2f %.2f %.2f", x, y, z);
```

Here, the values of variables `x`, `y`, and `z` are included in the formatted result. The format pattern includes each of the values (the first `%.2f` includes the value of variable `x`, the second `%.2f` includes the value of variable `y`, etc). The format pattern includes spaces between the format specifiers (between the `%.2f` parts), and thus the formatted values include spaces between them as well. When simulated, this could give the following output:

```
1.50 0.00 -3.57
2.34 3.75 5.78
4.71 12345.34 -3.12
-3.25 1.25 99.20
```

Due to some larger values being mixed with shorter values, as well as due to having both positive and negative values, the output is not so easy to read. This can be solved by using the following CIF specification instead:

```
print fmt("%10.2f %10.2f %10.2f", x, y, z);
```

By including a *width* of `10`, the text is now nicely formatted into columns of ten characters wide:

```
---------- ---------- ----------
      1.50       0.00      -3.57
      2.34       3.75       5.78
      4.71   12345.34      -3.12
     -3.25       1.25      99.20
```

The first line is not actual output, but is included to make it easier to see the columns. Not only is the output now nicely put in columns, but due to the exactly two digits after the dot (`.`), the dots are now also nicely aligned.

Large numbers

Consider the following CIF specification:

```
print fmt("%.2f", time);
```

When simulated, this could give the following output:

```
0.00
1.34
5000.23
```

```
2147185402.17
```

As the values of variable `time` get larger and larger, their textual representations become longer and longer, and it becomes more and more difficult to see exactly how large the values are. The following alternative CIF specification solves this problem:

```
print fmt("%,.2f", time);
```

The addition of the comma (`,`) *flag* ensures that we get the following output instead:

```
0.00
1.34
5,000.23
2,147,185,402.17
```

That is, a comma is used in the result as *thousand separator*, making it easier to see that the value of variable `time` is just over two billion, rather than for instance just over 200 million.

Left alignment

Earlier, we put values in columns. By default, if a *width* is used, text is right-justified (aligned to the right). We can also justify it to the left, as in this CIF specification:

```
print fmt("%-10.2d %-10.2d %-10.2d", g, h, i);
```

Here the `%d` specifier is used instead of the `%f` specifier. The `%d` specifier can only be used for integer values, while the `%f` specifier can only be used for real values. Besides a different specifier, the minus (`-`) flag is added, and different variables are used as values. After the changes, the CIF specification could result in the following output:

```
---------- ---------- ----------
184        3675       2
19350      29         -2956
-17        -964563    235
2946       567        -25072563
```

Once again, the first line is not actual output, but is included to make it easier to see the columns. Observe how the addition of the `-` flag resulted in the right-justified output being changed to left-justified (aligned to the left) output. The presence of negative numbers makes that the left-most digits of the columns are not nicely aligned. The following CIF specification solves this:

```
print fmt("%-+10.2d %-+10.2d %-+10.2d", x, y, z);
```

The addition of the plus (+) flag means that for non-negative numbers, a plus (+) character is always included:

```
---------- ---------- ----------
+184       +3675      +2
+19350     +29        -2956
-17        -964563    +235
+2946      +567       -25072563
```

If preferred, a space can be used instead of a plus, resulting in the following CIF specification:

```
print fmt("%- 10.2d %- 10.2d %- 10.2d", x, y, z);
```

and the following output:

```
---------- ---------- ----------
 184        3675       2
 19350      29        -2956
-17        -964563     235
 2946       567       -25072563
```

More formatting

In this lesson, we've seen a few forms of text formatting using the `fmt` function. However, CIF supports various other specifiers, besides the `%s`, `%f`, and `%d` specifiers that were used in this lesson. All those specifiers support various flags, widths, and precisions, and allow them to be combined in various ways. For the complete details of text formatting, see the reference manual's text formatting details documentation.

2.14. Synthesis

2.14.1. Supervisory controller synthesis

Supervisory controller synthesis (or supervisor synthesis, or just synthesis) is a generative technique, where one derives a supervisory controller from a collection of plants and requirements. Synthesis is the main ingredient of the synthesis-based engineering approach to develop supervisory controllers. Synthesis allows to focus on the *what*, i.e. which requirements should hold, rather than on the *how*, i.e. how to implement this in a controller.

The *plants* describe capabilities or behavior of a physical system 'as is', without any integrated control. They represent the available behavior of the uncontrolled system. *Requirements* model (a part of) the functions a system is supposed to perform. They represents behavior that is allowed in the controlled system, or more precisely, they specify the behavior that is not allowed in the controlled system. In other words, requirements restrict the behavior of the plants, to ensure that

only the desired behavior remains. The goal of supervisory controller synthesis is to compute a supervisory controller (or *supervisor*) that enforces the requirements, assuming the behavior of the plants, additionally preventing deadlock and livelock, and without restricting the system any further than is required.

CIF has several features that are used solely of modeling systems for the purpose of supervisory controller synthesis. These concepts are explained in upcoming lessons:

- Automaton kinds
- Invariant kinds
- Event controllability
- Marker predicates
- Reachability requirement annotations (as part of the 'annotations' lessons category)

When a model is not used for supervisory controller synthesis, e.g. for simulation, these concepts are usually ignored.

For an example of how these concepts can be used for modeling of a concrete example, see the synthesis-based engineering in practice example section.

2.14.2. Automaton kinds

For supervisory controller synthesis, different kinds of automata are treated in different ways. Regular automata, as used in the language tutorial so far, are specified using only the `automaton` keyword. Regular automata do not specify a kind for supervisory controller synthesis, and are therefore sometimes also referred to as kindless automata. Synthesis tools typically require knowledge about the purpose of each of the automata, and therefore don't support regular automata.

For supervisory controller synthesis, three different kinds of automata are available: plant automata, requirement automata, and supervisor automata. These automata are identical to regular automata, except for the keywords used to declare their intent. The `automaton` keyword is preceded or replaced by the `plant`, `requirement`, or `supervisor` keyword respectively.

For instance, the following are two alternative ways to model the same plant automaton:

```
// Plant automaton, long form.
plant automaton lamp:
  event turn_on, turn_off;

  location off:
    initial;
    edge turn_on goto on;

  location on:
    edge turn_off goto off;
```

```
end

// Plant automaton, short form.
plant lamp:
  event turn_on, turn_off;

  location off:
    initial;
    edge turn_on goto on;

  location on:
    edge turn_off goto off;
end
```

2.14.3. Invariant kinds

Similar to automata, different kinds of invariants are treated in different ways for supervisory controller synthesis. Invariants can be declared as plant, requirement, or supervisor invariants, by preceding or replacing the `invariant` keyword with a `plant`, `requirement`, or `supervisor` keyword respectively.

For instance, consider the following CIF specification:

```
invariant inv1: lamp.on or lamp.off;      // Regular/kindless invariant.

plant lamp:
  event turn_on, turn_off;

  requirement invariant req1: not off;    // Requirement invariant. Long form.
  requirement req2: not off;              // Requirement invariant. Short form.

  location off:
    initial;
    edge turn_on goto on;

  location on:
    edge turn_off goto off;
end
```

The `inv1` invariant does not specify an additional supervisory kind keyword. It is therefore a regular invariant, also called a kindless invariant. The `req1` invariant, in the `lamp` automaton, explicitly specifies a `requirement` kind and is therefore a requirement invariant. The `req2` invariant is identical to the `req1` invariant, except that the `invariant` keyword has been omitted. It is also a requirement invariant, but written in a shorter form.

There are two forms of invariants: state invariants and state/event exclusion invariants. Both forms can be used for synthesis. For instance, consider the following example of two counters:

```
plant def counter():
  controllable c_increment, c_decrement;
  disc int[0..10] count = 0;

  location:
    initial;
    marked;
    edge c_increment do count := count + 1;
    edge c_decrement do count := count - 1;
end

C1: counter();
C2: counter();

plant invariant C1.count + C2.count <= 15;
requirement C2.c_increment needs C1.count > 0;
```

Each counter can be incremented and decremented. The plant invariant specifies that the counts of both counters summed together can never exceed 15. The requirement invariant specifies that counter `C2` may only be incremented if the `count` of `C1` is non-zero. Note that data-based synthesis automatically ensures that all integer-typed variables are kept within range. Therefore, it is not strictly necessary to add requirements or guards to do so.

2.14.4. Event controllability

Supervisory controller synthesis distinguishes two kinds of events: controllable events and uncontrollable events.

Uncontrollable events are events that a supervisor can not prohibit, and are usually related to sensors. A typical example are events that indicate that a button has been pushed or released. A button is essentially a sensor that is on if the user pushes the button, and off if the user doesn't push it. The supervisor can not prevent a user from physically pushing or releasing a button, and can also not determine when the user pushes or releases it. The supervisor thus also can not prevent the events from occurring.

Another example is an event that indicates that a moving part has reached its outer position (limit sensor turns on). If the part reaches its outer position, the event will occur. The supervisor can not control the sensor, as it is physically linked to the position of the moving part.

Controllable events may be restricted by a supervisor, and are usually related to actuators. Typical examples are events used to turn a motor on or off, to turn a lamp on or off, or to change the direction of movement.

Even though a supervisor may not be able to control a limit sensor directly, and thus restrict the corresponding events, it may be able to influence it indirectly. For instance, a motor may be available that makes it possible for the part to move. That motor may then be controlled using controllable on and off events. Stopping the motor then makes the part stop moving, ensuring that

the part never reaches its outer position, indirectly preventing the limit sensor from turning on, and the corresponding event from happening.

Events and channels in CIF are declared using the `event` keyword. For controllable events, the `event` keyword may be preceded or replaced by the `controllable` keyword. Similarly, for uncontrollable events, the `event` keyword may be preceded or replaced by the `uncontrollable` keyword. Similar to event declarations, for event parameters the `event` keyword may by preceded or replaced by the `controllable` or `uncontrollable` keyword, for controllable and uncontrollable event parameters respectively.

As a convention, controllable events are given names that start with `c_`, and uncontrollable events are given names that start with `u_`. This allows them to be highlighted in different colors.

For instance, consider the following CIF specification:

```
controllable c_on, c_off;
uncontrollable u_pushed, u_released;
```

2.14.5. Marker predicates

Marking is very weak form of liveness, and is used by supervisory controller synthesis to prevent livelocks, to ensure progress. It is also used to prevent deadlocks. A supervisor per definition ensures that a marked state can always be reached, for the entire system. A system is marked if all its automata are marked. An automaton is marked if its active location is a marked location. In literature, marked location are also called *safe* locations. Note that deadlock in marked states is not prevented, as a marker state (the state itself) can be reached by not taking any transition. In literature, marked deadlock states are also called *final* states.

Marking can be specified using *marker predicates*. Locations can be marked using the `marked` keyword, as follows:

```
plant lamp:
  controllable c_turn_on, c_turn_off;

  location off:
    initial;
    marked;
    edge c_turn_on goto on;

  location on:
    edge c_turn_off goto off;
end
```

In this example location `off` is both initial and marked, while location `on` is neither initial nor marked.

Marker predicates are very similar to initialization predicates, as by default, locations are not marked, similar to how by default locations are not initial. Marker predicates can be specified in locations, automata, groups, and the top level of the specification (which can be seen as a group), similar to initialization predicates.

The following specification shows an example of a variable that is only marked for a certain value:

```
plant counter:
  controllable c_increment, c_decrement;
  disc int[0..10] count = 0;

  marked count = 0;

  location:
    initial;
    marked;
    edge c_increment do count := count + 1;
    edge c_decrement do count := count - 1;
end
```

Variable `count` of plant automaton `counter` is only marked if it has value zero. That is, the entire system can only be marked if `count` is zero. Supervisor synthesis will ensure that it is always possible to get back to a state where `count` is zero.

2.15. Annotations

2.15.1. Annotations

Annotations can be added to elements of a CIF specification, to annotate them with extra information. Annotations are a form of structured data, and the CIF type checker checks their validity. Tools that take CIF specifications as input can process the annotations that are attached to elements of the specification, and use the supplied information.

Annotations differ from comments. Comments can also be used to add additional information to a specification. However, comments are completely free form, as you can write anything in them. They can even be used to (temporarily) comment out some parts of the specification. Furthermore, comments are purely for the benefit of the modeler. They are never interpreted by a tool.

Annotations have two main purposes:

- Annotations allow for extensibility, as anyone can add annotations, whether it is general purpose, company-specific, or even for private use.
- Annotations allow to test out language extensions before fully integrating them into the language.

Annotations can be added to most elements of CIF specifications that have a name (like components and variables), or can have a name (like locations and invariants), as well as some other elements (like edges).

The following built-in annotations are bundled with CIF:

- `controller:properties`
- `doc`
- `partial`
- `requirement:reachable`
- `state`

Annotations are an extension mechanism, and anyone can define and register their own annotations. Different CIF installations may therefore have different registered annotations. A CIF specification may thus have a warning in one installation, if the annotation is for instance not registered in that installation, while in another installation the annotation does not have a warning, as there it is registered. Different people could define the same annotation in different ways, such that their own installations interpret that annotation differently and impose different constraints for it. Hence, a specification with such annotations may be valid in one installation, and invalid in another.

For more detailed information on annotations, see the reference manual.

2.15.2. Controller properties annotations

A controller properties annotation can be added to a CIF specification that represents a controller that satisfies certain properties. The annotation should not be manually added to specifications. Instead, it should be added by tools such as the CIF controller properties checker tool, once the tool has checked that the properties hold for the specification. For instance, such a tool may add the following to a CIF specification that has bounded response:

```
@@controller:properties(boundedResponse: true, uncontrollablesBound: 2,
controllablesBound: 5)
```

For more detailed information on controller properties annotations, see the reference manual.

2.15.3. Documentation annotations

Documentation can be added to all elements of the specification that support annotations:

```
@doc("The lower limit sensor of the elevator.")
@doc("Safety sensor.")
input bool ElevatorDownSensor;
```

The documentation may span multiple lines, by using explicit newline characters:

```
@doc(
    "doc with multiple\n" +
    "lines of\n" +
    "text"
)
input bool i;
```

This also may be abbreviated as follows, by using multiple arguments:

```
@doc(
    "doc with multiple",
    "lines of",
    "text",
)
input bool i;
```

It is also possible to use for instance constants from the model in the documentation text:

```
const int MAX_NR_OF_PRODUCTS = 3;

@doc(fmt("Sensor 1/%d.", MAX_NR_OF_PRODUCTS))
input bool s1;
```

This documentation can then be used by CIF tools in various ways. For instance, if the CIF code generator is used to generate Java code for the above example, then the documentation is included in the generated code:

```
/**
 * Input variable "ElevatorDownSensor".
 *
 * <p>
 * The lower limit sensor of the elevator.
 * </p>
 *
 * <p>
 * Safety sensor.
 * </p>
 */
public boolean ElevatorDownSensor;
```

For more detailed information on documentation annotations, see the reference manual.

2.15.4. Partial specification annotations

CIF supports defining library models that can be imported by multiple different other models, to allow reuse and reduce duplication among models. For instance, consider the following CIF model

that defines some events that are to be used in both a plant model a simulation model:

```
controllable c_start;
controllable c_stop;
```

A library model is typically not a complete specification, but rather only a partial specification. It contains the common parts, while other models that import the library model provide the differing parts.

The CIF type checker checks models for common problems, such as unused events. For partial specifications this may lead to false positives. For instance, in the model above, the `c_start` and `c_stop` events are not used in the library model itself. The plant and simulation model that import it likely do use the events, ensuring that the events are used in the full specification. Since the type checker does not know where a library will be imported, it may incorrectly warn about such unused events.

To indicate to the type checker that a model is a partial specification, you can add the `@@partial` annotation. This indicates to the type checker that it shouldn't report warnings that are typically false positives in partial specifications, such as unused events:

```
@@partial

controllable c_start;
controllable c_stop;
```

For more detailed information on partial specification annotations, see the reference manual.

2.15.5. Reachability requirement annotations

CIF supports marker predicates that allow to specify marked states. Supervisory controller synthesis then ensures that from every state in the controlled system, a marked state can always be reached. Marking serves as a means to specify a form of liveness requirements. However, marking is a very weak form of liveness; in every state only a single marked state needs to be reachable. In many cases, you may want to express stronger liveness requirements. You can use reachability requirements for this.

Marking is not always sufficient

To show why marking alone is not always sufficient, consider a bridge that may be open or closed, modeled as an automaton with two locations:

```
plant bridge:
  controllable c_open, c_close;

  location open:
    initial;
    marked;
```

```
    edge c_close goto closed;

  location closed:
    edge c_open goto open;
end
```

We could mark the location where the bridge is open, to ensure it can always be opened. However, then synthesis could determine that it needs to forbid closing the bridge (event `c_close`), for instance due to some other requirement that we wrote. The automaton would then remain in its marked `open` location. The supervisor would not be empty, so you may assume you got the desired supervisor. You could find the problem that the bridge is always open and can never close by means of simulation. However, for large models with many possible simulation scenarios, such problems could be hard to find, especially if they concern behaviors that do not occur very often.

Alternatively, we could mark the location where the bridge is closed. However, then synthesis could similarly decide it needs to forbid opening the bridge.

Yet another alternative would be to mark both locations. However, synthesis then only needs to ensure that one of those locations can always be reached. It could still block either opening or closing the bridge.

Reachability requirements

What we want for this model, is that you can always open the bridge, but also always close the bridge. Both must always be possible. You can specify that using reachability requirements, as follows:

```
@requirement:reachable(open)
@requirement:reachable(closed)
plant bridge:
  controllable c_open, c_close;

  location open:
    initial;
    marked;
    edge c_close goto closed;

  location closed:
    edge c_open goto open;
end
```

The plant automaton now has two reachability requirements. The first requirement indicates that a state where `open` is the current location of `bridge` must always be reachable. The second requirement indicates that a state where `closed` is the current location of `bridge` must always be reachable.

Together these two requirements will ensure that the bridge can always be opened and always be closed. That is, synthesis will ensure that for every state in the controlled system, it is possible to

reach a state where `open` holds and it is possible to reach a state where `closed` holds. If that is somehow not possible, for instance due to other parts of the specification, such as conflicting requirements, then synthesis results in an 'empty supervisor' error.

Reachability requirements may be specified in specifications, on components (automata and groups), on component definitions and on component instantiations. This allows to specify them as locally as possible. This makes it easier to see to what part of the specification they relate. It also makes it easier to specify them, as the variables, locations, and so on, of the local scope can be referred to directly. Consider for instance the following alternative to the specification above, where the reachability requirements are now specified in the root of the specification, rather than on the automaton:

```
@@requirement:reachable(system.bridge.open)
@@requirement:reachable(system.bridge.closed)

group system:
  plant bridge:
    controllable c_open, c_close;

    location open:
      initial;
      marked;
      edge c_close goto closed;

    location closed:
      edge c_open goto open;
  end
end
```

In general, each reachability requirement specifies a goal predicate, such as `open` or `closed` in the example above. For the requirements in this example, synthesis ensures that every state in the controlled system can reach a state that satisfies that predicate.

A reachability requirement that only specifies a goal predicate is called a basic reachability requirement. It can be extended with various additional arguments as explained below.

Both basic and extended reachability requirements ensure that a path to the goal remains available wherever the requirement applies, subject to any restrictions specified by its arguments. They do not enforce that such a path is actually taken.

Marking vs basic reachability requirements

Basic reachability requirements are very similar to marking. Synthesis will ensure that from every state in the controlled system, always a marked state can be reached. Similarly, for a basic reachability requirement, synthesis will ensure that from every state in the controlled system a state can always be reached that satisfies the requirement's goal predicate. In the bridge example above, the basic reachability requirement `@requirement:reachable(open)` is superfluous, as it is already ensured by marking.

There are a few key differences between marking and basic reachability requirements. The first is that you can specify as many basic reachability requirements as you want, and that each defines an independent set of goal states, at least one of which must be reachable from every state in the controlled system. For marking there is only one such set of states.

The second is that marking needs to be ensured globally. A specification is marked if each automaton is marked, and an automaton is marked if its current location is marked. Marking must thus be considered for every automaton. Basic reachability requirements can be specified locally for a single automaton, without having to think about any of the other automata. The combination of marking and basic reachability requirements allows you to benefit from the guarantees of global completeness as well as the ease of local specification.

Note that marking is mandatory for supervisory controller synthesis, as without marking you will always get an empty supervisor. Basic reachability requirements on the other hand are optional.

Conditional reachability with `from`

The optional `from` argument defines conditional reachability. It restricts the source states to which the requirement applies:

```
@requirement:reachable(closed, from: open)
```

This requirement only requires states where `open` holds to be able to reach a state where `closed` holds. States that do not satisfy `open` are not affected by this requirement.

On its own, the requirement does not ensure that any `open` state remains reachable in the controlled system. For example, if the bridge is initially closed and no other requirement demands that it can open, synthesis could prevent it from opening, leaving no source states to which this requirement applies. If opening must also remain possible, the basic reachability requirement `@requirement:reachable(open)` can be added. This ensures that from every state in the controlled system, at least one state where `open` holds remains reachable.

Restricted reachability with `avoid` or `stay`

The optional `avoid` and `stay` arguments define restricted reachability by limiting the states that may occur on the path to the goal. They also determine the states to which the requirement applies.

The annotation `@requirement:reachable(goal, avoid: avoid_pred)` requires reaching `goal` without visiting states that satisfy `avoid_pred`. The avoid states are not allowed to occur anywhere along the path to the goal, including the starting and goal states. States that already satisfy `avoid_pred` are not affected by this reachability requirement.

The annotation `@requirement:reachable(goal, stay: stay_pred)` requires reaching `goal` while staying in states that satisfy `stay_pred`. For states to which the requirement applies, this means every state on the path to the goal, including the starting and goal states, must satisfy `stay_pred`. States that already do not satisfy `stay_pred` are not affected by this reachability requirement.

The `avoid` and `stay` arguments express the same kind of path restriction in inverse forms: `avoid: pred` is equivalent to `stay: not pred`, and `stay: pred` is equivalent to `avoid: not pred`. The arguments

are mutually exclusive, so use the form that best describes the intended behavior and avoids unnecessary logical negations.

The following annotations illustrate restrictions on paths for opening and closing a bridge:

```
@requirement:reachable(open, avoid: maintenance)
@requirement:reachable(closed, stay: operational)
@requirement:reachable(closed, from: open, stay: operational)
plant bridge:
  ...
end
```

The first requirement indicates that `open` must remain reachable from states outside `maintenance`, without visiting a maintenance state. The second one indicates that `closed` must remain reachable from operational states along a path of operational states. The third one combines conditional and restricted reachability: from states satisfying both `open` and `operational`, `closed` must remain reachable along an operational path.

Bounded reachability with `bound`

The optional `bound` argument defines bounded reachability by limiting the maximum number of transitions on the path to the goal:

```
@requirement:reachable(closed, bound: 3)
@requirement:reachable(open, from: closed, bound: 2)
@requirement:reachable(open, from: closed, stay: operational, bound: 2)
plant bridge:
  ...
end
```

The first requirement requires the goal to be reachable in at most three transitions from every state in the controlled system. The second requirement combines conditional and bounded reachability, so only states satisfying `closed` must be able to reach `open` within two transitions. The third also specifies `stay: operational`, so states satisfying both `closed` and `operational` must be able to reach `open` within two transitions along operational states. Conditional, restricted and bounded reachability can be combined in this way.

For `bound: 0`, a state satisfies the requirement only if it already satisfies the goal predicate, unless it is not affected by the requirement due to a `from`, `avoid` or `stay` argument.

The bound counts discrete transitions, not elapsed time. Choosing a bound that is too small may disable behavior or result in an empty supervisor. For guidance on choosing reachability requirements and bounds, see modeling reachability requirements in practice.

For more detailed information on reachability requirement annotations, see the reference manual.

2.15.6. State annotations

A state annotation adds state information to a location in an automaton. States consist of the current locations of automata and the current values of variables. The typical use of state annotations is for generated models that contain an automaton representing a state space of another model, where each location of that automaton represents a state of the state space. For instance, the CIF explorer may generate CIF models with state spaces represented as automata. For the example from the lesson on synchronizing events, it may generate the following model:

```
group producer:
  event produce;
  event provide;
end

group consumer:
  event consume;
end

automaton statespace:
  alphabet producer.produce, producer.provide, consumer.consume;

  @state(consumer: "idle", producer: "producing")
  location loc1:
    initial;
    edge producer.produce goto loc2;

  @state(consumer: "idle", producer: "idle")
  location loc2:
    edge producer.provide goto loc3;

  @state(consumer: "consuming", producer: "producing")
  location loc3:
    edge producer.produce goto loc4;
    edge consumer.consume goto loc1;

  @state(consumer: "consuming", producer: "idle")
  location loc4:
    edge consumer.consume goto loc2;
end
```

This state space consists of four states, each represented by a location of the `statespace` automaton. For instance, the first state is represented by location `loc1`, which corresponds to the state where the `consumer` is `idle` and the `producer` is `producing`.

A location may have multiple state annotations in case it represents multiple states. For instance, the event-based DFA minimization tool may merge locations representing different states to a single location representing multiple states.

For more detailed information on state annotations, see the reference manual.

Chapter 3. Language reference

The documentation in this language reference manual serves as reference. It is not intended to be read from beginning to end, but more to look up specific information when needed. The reference documentation generally provides detailed and complete information, but this information may not always be presented in the most accessible way.

The information is divided into categories. The following categories and information is part of the language reference documentation:

Syntax

- Lexical syntax
- Grammar
- CIF XML files

SVG visualization and interaction

- CIF/SVG declarations
  - SVG file declarations
  - CIF/SVG copy declarations
  - CIF/SVG move declarations
  - CIF/SVG output mappings
  - CIF/SVG input mappings

Text

- Print output
  - Print declarations
  - Print file declarations
- Text formatting details

Annotations

- Annotations
- Built-in annotations
  - Controller properties annotations
  - Documentation annotations
  - Reachability requirement annotations
  - State annotations

3.1. Syntax

3.1.1. Lexical syntax

CIF files are textual files. While CIF files can have any file extension, by convention a `.cif` file extension is used.

This page describes the CIF lexical syntax.

Characters

CIF files may only contain ASCII characters (0 through 127).

The encoding of CIF files is assumed to be UTF-8. If a CIF file is actually encoded using a different encoding, error messages produced while reading it may indicate the wrong characters.

Keywords

Language keywords

```
alg            disc       group       post          switch
alphabet       dist       id          pre           tau
any            do         if          print         text
attr           edge       import      printfile     time
automaton      elif       initial     real          to
bool           else       input       requirement   true
break          end        int         return        tuple
case           enum       invariant   self          type
const          equation   list        set           uncontrollable
cont           event      location    string        urgent
continue       false      marked      supervisor    value
controllable   file       monitor     svgcopy       void
def            final      namespace   svgfile       when
der            for        needs       svgin         while
dict           func       now         svgmove
disables       goto       plant       svgout
```

Trigonometric functions

```
acosh   asin    cosh   sin
acos    atanh   cos    tanh
asinh   atan    sinh   tan
```

General functions

```
abs    empty   ln    pop     sign
cbrt   exp     log   pow     size
ceil   floor   max   round   sqrt
del    fmt     min   scale
```

Distributions

```
bernoulli   erlang        lognormal   triangle
beta        exponential   normal      uniform
binomial    gamma         poisson     weibull
constant    geometric     random
```

Expression operators

```
and   mod   sample
div   not   sub
in    or
```

Terminals

Besides the keyword terminals listed above, CIF features several other terminals:

`IDENTIFIERTK`

An identifier. Defined by the regular expression: `[$]?[a-zA-Z_][a-zA-Z0-9_]*`. They thus consist of letters, numbers and underscore characters (`_`). Identifiers may not start with a numeric digit. Keywords take priority over identifiers. To use a keyword as an identifier, prefix it with a `$` character. The `$` is not part of the identifier name.

Examples:

```
apple       // identifier
bear        // identifier
int         // keyword
$int        // identifier 'int' (override keyword priority with $)
```

`RELATIVENAMETK`

A relative name. Defined by the regular expression: `[$]?[a-zA-Z_][a-zA-Z0-9_]*(\.[$]?[a-zA-Z_][a-zA-Z0-9_]*)+`. It thus consists of two or more `IDENTIFIERTK` joined together with periods (`.`).

Examples:

```
some_automaton.some_location
```

ABSOLUTENAMETK

An absolute name. Absolute names can be used to refer to objects that are otherwise hidden. It represents an absolute name from the root of the current scope.

Defined by the regular expression: `\.[$]?[a-zA-Z_][a-zA-Z0-9_]*(\.[$]?[a-zA-Z_][a-zA-Z0-9_]*)*`. It starts with a period (`.`), and then follows an `IDENTIFIER` or `RELATIVENAMETK`.

Examples:

```
.some_event
.some_group.some_event
```

ROOTNAMETK

A root name. Absolute names can be used to refer to objects that are otherwise hidden. It represents an absolute name from the root of the current specification.

Defined by the regular expression: `\^[$]?[a-zA-Z_][a-zA-Z0-9_]*(\.[$]?[a-zA-Z_][a-zA-Z0-9_]*)*`. It starts with a circumflex accent (`^`), and then follows an `IDENTIFIER` or `RELATIVENAMETK`.

Examples:

```
^some_group.some_event
```

REGULAR_ANNOTATION_NAMETK

A (regular) annotation name. Regular annotation names are used to refer to annotations, when they are used to annotate most elements of CIF specifications.

Defined by the regular expression: `@[a-zA-Z_][a-zA-Z0-9_]*(:[a-zA-Z_][a-zA-Z0-9_]*)*`. It starts with an at sign (`@`), and then follow one or more `IDENTIFIER` terminals, separated by colons (`:`). Within annotation names, the identifiers are never escaped (no `$`). The at sign is only used to indicate that an annotation name follows, but it is not part of the annotation name itself.

Examples:

```
@doc
@plc:input
```

DOUBLE_ANNOTATION_NAMETK

A double at-sign annotation name. Double at-sign annotation names are used to refer to annotations, when they are used to annotate certain elements of CIF specifications, such as the entire specification. Double at-sign annotation names are identical to regular annotation names, but starts with two at signs (`@@`).

Defined by the regular expression: `@@[a-zA-Z_][a-zA-Z0-9_]*(:[a-zA-Z_][a-zA-Z0-9_]*)*`. It starts with two at signs (`@@`), and then follow one or more `IDENTIFIER` terminals, separated by colons (`:`). Within annotation names, the identifiers are never escaped (no `$`). The at signs are only used to indicate that an annotation name follows, but they are not part of the annotation name itself.

Examples:

```
@@doc
@@plc:input
```

NUMBERTK

An integer literal. Defined by the regular expression: `0|[1-9][0-9]*`. Integers thus consist of numeric digits. Only for the number `0` may an integer literal start with `0`. E.g. `02` is invalid.

Examples:

```
0
1
123
```

REALTK

A real literal. Defined by the regular expression: `(0|[1-9][0-9]*)(\.[0-9]+|(\.[0-9]+)?[eE][\-\+]?[0-9]+)`. Simple double literals consist of an integer literal followed by a period (`.`) and some numeric digits. Double literals using scientific notation start with either an integer literal or a simple double literal. They then contain either an `e` or `E`, followed by the exponent. The exponent consists of numeric digits, optionally preceded by `+` or `-`.

Examples:

```
0.0
1e5
1E+03
1.05e-78
```

`STRINGTK`

A string literal. Defined by the regular expression: `\"([^\\\"\n]|\\[nt\\\"])*\"`. String literals are enclosed in double quotes (`"`). String literals must be on a single line and must thus not include new line characters (`\n`, Unicode U+0A). To include a double quote (`"`) in a string literal, it must be escaped as `\"`. Since a backslash (`\`) serves as escape character, to include a backslash in a string literal it must be escaped as `\\`. To include a tab character in a string literal, use `\t`. To include a newline in a string literal, use `\n`.

Examples:

```
"hello world"
"first line\nsecond line"
```

Whitespace

CIF supports spaces, tabs, and new line characters as whitespace. Whitespace is ignored (except in string literals), but can be used to separate tokens as well as for layout purposes. The use of tab characters is allowed, but should be avoided if possible, as layout will be different for text editors with different tab settings. You may generally format a CIF script as you see fit, and start on a new line when desired.

Examples:

```
// Normal layout.
int x = 5;

// Alternative layout.
int
  x    =
    5
  ;
```

Comments

CIF features two types of comments. Single line comments start with `//` and end at end of the line. Multi line comments start with `/*` and end at `*/`. Comments are ignored.

Examples:

```
int x = 5; // Single line comment.

int /* some comment */ x = /* some
  more comments
  and some more
```

```
 end of the multi line comment */ 5;
```

3.1.2. Grammar

Below, the grammar of the CIF language is shown, in a form closely resembling Backus-Naur Form (BNF). The `Specification` non-terminal is the start symbol of the grammar. Text between quotes are terminals. Names using only upper case letters are defined in the lexical syntax. The remaining names are the non-terminals of the grammar.

```
Specification : OptDoubleAnnos GroupBody
              ;

GroupBody : OptGroupDecls
          ;

AutomatonBody : OptAutDecls Locations OptIoDecls
              ;

OptGroupDecls : /* empty */
              | OptGroupDecls GroupDecl
              ;

GroupDecl : Decl
          | "import" Imports ";"
          | "namespace" IDENTIFIERTK ";"
          | "namespace" RELATIVENAMETK ";"
          |       "func" Types Identifier FuncParams ":" FuncBody
          | Annos "func" Types Identifier FuncParams ":" FuncBody
          |       Identifier ":" Name CompInstArgs ";"
          | Annos Identifier ":" Name CompInstArgs ";"
          |       "group"               "def" Identifier CompDefParms ":" GroupBody
"end"
          | Annos "group"               "def" Identifier CompDefParms ":" GroupBody
"end"
          |       OptSupKind "automaton" "def" Identifier CompDefParms ":"
AutomatonBody "end"
          | Annos OptSupKind "automaton" "def" Identifier CompDefParms ":"
AutomatonBody "end"
          |          SupKind             "def" Identifier CompDefParms ":"
AutomatonBody "end"
          | Annos    SupKind             "def" Identifier CompDefParms ":"
AutomatonBody "end"
          |       "group"                     Identifier             ":" GroupBody
"end"
          | Annos "group"                     Identifier             ":" GroupBody
"end"
          |       OptSupKind "automaton"      Identifier             ":"
```

```
AutomatonBody "end"
          | Annos OptSupKind "automaton"       Identifier             ":"
AutomatonBody "end"
          |          SupKind                   Identifier             ":"
AutomatonBody "end"
          | Annos    SupKind                   Identifier             ":"
AutomatonBody "end"
          ;

OptAutDecls : /* empty */
            | OptAutDecls AutDecl
            ;

AutDecl : Decl
        | "alphabet" Events ";"
        | "alphabet" ";"
        | "monitor" Events ";"
        | "monitor" ";"
        |       "disc" Type DiscDecls ";"
        | Annos "disc" Type DiscDecls ";"
        ;

Decl :       "type" TypeDefs ";"
     | Annos "type" TypeDefs ";"
     |       "enum" Identifier "=" AnnotatedIdentifiers ";"
     | Annos "enum" Identifier "=" AnnotatedIdentifiers ";"
     |       OptControllability "event"           Identifiers ";"
     | Annos OptControllability "event"           Identifiers ";"
     |       OptControllability "event" EventType Identifiers ";"
     | Annos OptControllability "event" EventType Identifiers ";"
     |          Controllability                    Identifiers ";"
     | Annos    Controllability                    Identifiers ";"
     |          Controllability         EventType Identifiers ";"
     | Annos    Controllability         EventType Identifiers ";"
     |       "const" Type ConstantDefs ";"
     | Annos "const" Type ConstantDefs ";"
     |       "alg" Type AlgVarsDefs ";"
     | Annos "alg" Type AlgVarsDefs ";"
     |       "input" Type Identifiers ";"
     | Annos "input" Type Identifiers ";"
     |       "cont" ContDecls ";"
     | Annos "cont" ContDecls ";"
     | "equation" Equations ";"
     | "initial" Expressions ";"
     |       InvariantDecls
     | Annos InvariantDecls
     | "marked" Expressions ";"
     | IoDecl
     ;

Identifiers : Identifier
```

```
            | Identifiers "," Identifier
            ;

Identifier : IDENTIFIERTK
           ;

AnnotatedIdentifiers : AnnotatedIdentifier
                     | AnnotatedIdentifiers "," AnnotatedIdentifier
                     ;

AnnotatedIdentifier : OptAnnos IDENTIFIERTK
                    ;

Imports : StringToken
        | Imports "," StringToken
        ;

StringToken : STRINGTK
            ;

TypeDefs : Identifier "=" Type
         | TypeDefs "," Identifier "=" Type
         ;

ConstantDefs : Identifier "=" Expression
             | ConstantDefs "," Identifier "=" Expression
             ;

AlgVarsDefs : Identifier
            | Identifier "=" Expression
            | AlgVarsDefs "," Identifier
            | AlgVarsDefs "," Identifier "=" Expression
            ;

FuncParams : "(" ")"
           | "(" FuncParamDecls ")"
           ;

FuncParamDecls : FuncParamDecl
               | FuncParamDecls ";" FuncParamDecl
               ;

FuncParamDecl : OptAnnos Type Identifiers
              ;

FuncBody : FuncVarDecls FuncStatements "end"
         | StringToken ";"
         ;

FuncVarDecls : /* empty */
             | FuncVarDecls       Type FuncVarDecl ";"
```

```
             | FuncVarDecls Annos Type FuncVarDecl ";"
             ;

FuncVarDecl : Identifier
            | Identifier "=" Expression
            | FuncVarDecl "," Identifier
            | FuncVarDecl "," Identifier "=" Expression
            ;

FuncStatements : FuncStatement
               | FuncStatements FuncStatement
               ;

FuncStatement : Addressables ":=" Expressions ";"
              | "if" Expressions ":" FuncStatements
                OptElifFuncStats OptElseFuncStat "end"
              | "while" Expressions ":" FuncStatements "end"
              | "break" ";"
              | "continue" ";"
              | "return" Expressions ";"
              ;

OptElifFuncStats : /* empty */
                 | OptElifFuncStats "elif" Expressions ":" FuncStatements
                 ;

OptElseFuncStat : /* empty */
                | "else" FuncStatements
                ;

Events : Name
       | Events "," Name
       ;

CoreEdge : EdgeEvents         OptEdgeGuard OptEdgeUrgent OptEdgeUpdate
         | "when" Expressions              OptEdgeUrgent OptEdgeUpdate
         | "now"                                         OptEdgeUpdate
         | "do" Updates
         ;

OptEdgeGuard : /* empty */
             | "when" Expressions
             ;

OptEdgeUrgent : /* empty */
              | "now"
              ;

OptEdgeUpdate : /* empty */
              | "do" Updates
              ;
```

```
EdgeEvents : EdgeEvent
           | EdgeEvents "," EdgeEvent
           ;

EdgeEvent : "tau"
          | Name
          | Name "!"
          | Name "!" Expression
          | Name "?"
          ;

Locations : Location
          | Locations Location
          ;

Location :       "location" ";"
         | Annos "location" ";"
         |       "location" Identifier ";"
         | Annos "location" Identifier ";"
         |       "location" ":" LocationElements
         | Annos "location" ":" LocationElements
         |       "location" Identifier ":" LocationElements
         | Annos "location" Identifier ":" LocationElements
         ;

LocationElements : LocationElement
                 | LocationElements LocationElement
                 ;

LocationElement : "initial" ";"
                | "initial" Expressions ";"
                | OptDoubleAnnos InvariantDecls
                | "equation" Equations ";"
                | "marked" ";"
                | "marked" Expressions ";"
                | "urgent" ";"
                | OptDoubleAnnos "edge" CoreEdge ";"
                | OptDoubleAnnos "edge" CoreEdge "goto" Identifier ";"
                ;

CompInstArgs : "(" ")"
             | "(" Expressions ")"
             ;

CompDefParms : "(" ")"
             | "(" CompDefDecls ")"
             ;

CompDefDecls : CompDefDeclaration
             | CompDefDecls ";" CompDefDeclaration
```

```
            ;

CompDefDeclaration : OptControllability "event"           EventParamIds
                   | OptControllability "event" EventType EventParamIds
                   |    Controllability                   EventParamIds
                   |    Controllability         EventType EventParamIds
                   | Name Identifiers
                   | "location" Identifiers
                   | "input" Type Identifiers
                   | OptAnnos "alg" Type Identifiers
                   ;

EventParamIds : EventParamId
              | EventParamIds "," EventParamId
              ;

EventParamId : Identifier OptEventParamFlags
             ;

OptEventParamFlags : /* empty */
                   | OptEventParamFlags EventParamFlag
                   ;

EventParamFlag : "!"
               | "?"
               | "~"
               ;

DiscDecls : DiscDecl
          | DiscDecls "," DiscDecl
          ;

DiscDecl : Identifier
         | Identifier "in" "any"
         | Identifier "=" Expression
         | Identifier "in" "{" Expressions "}"
         ;

ContDecls : ContDecl
          | ContDecls "," ContDecl
          ;

ContDecl : Identifier OptDerivative
         | Identifier "=" Expression OptDerivative
         ;

OptDerivative : /* empty */
              | "der" Expression
              ;

Equations : Equation
```

```
          | Equations "," Equation
          ;

Equation : Identifier "'" "=" Expression
         | Identifier "=" Expression
         ;

InvariantDecls : OptSupKind "invariant" Invariants ";"
               |    SupKind              Invariants ";"
               ;

Invariants : Invariant
           | Invariants "," Invariant
           ;

Invariant :                Expression
          | Identifier ":" Expression
          |                Name                   "needs"    Expression
          | Identifier ":" Name                   "needs"    Expression
          |                NonEmptySetExpression "needs"    Expression
          |                Expression             "disables" Name
          | Identifier ":" Expression             "disables" Name
          |                Expression             "disables" NamesSet
          ;

NamesSet : "{" Names "}"
         ;

Names : Name
      | Names "," Name
      ;

Updates : Update
        | Updates "," Update
        ;

Update : Addressable ":=" Expression
       | "if" Expressions ":" Updates
         OptElifUpdates OptElseUpdate "end"
       ;

Addressables : Addressable
             | Addressables "," Addressable
             ;

Addressable : Name
            | Name Projections
            | "(" Addressable "," Addressables ")"
            ;

Projections : Projection
```

```
            | Projections Projection
            ;

Projection : "[" Expression "]"
           ;

OptElifUpdates : /* empty */
               | OptElifUpdates "elif" Expressions ":" Updates
               ;

OptElseUpdate : /* empty */
              | "else" Updates
              ;

OptSupKind : /* empty */
           | SupKind
           ;

OptControllability : /* empty */
                   | Controllability
                   ;

Controllability : "controllable"
                | "uncontrollable"
                ;

////////////////////////////////////////////////////////////////////////////

OptIoDecls : /* empty */
           | OptIoDecls IoDecl
           ;

IoDecl : SvgFile
       | SvgCopy
       | SvgMove
       | SvgOut
       | SvgIn
       | PrintFile
       | Print
       ;

SvgFile : "svgfile" StringToken ";"
        ;

OptSvgFile : /* empty */
           | "file" StringToken
           ;

SvgCopy : "svgcopy" "id" Expression OptSvgCopyPre OptSvgCopyPost OptSvgFile ";"
        ;
```

```
OptSvgCopyPre : /* empty */
              | "pre" Expression
              ;

OptSvgCopyPost : /* empty */
               | "post" Expression
               ;

SvgMove : "svgmove" "id" Expression "to" Expression "," Expression
          OptSvgFile ";"
        ;

SvgOut : "svgout" "id" Expression SvgAttr "value" Expression OptSvgFile
         ";"
       ;

SvgAttr : "attr" StringToken
        | "text"
        ;

SvgIn : "svgin" "id" Expression "event" SvgInEvent OptSvgFile ";"
      | "svgin" "id" Expression "do"    Updates    OptSvgFile ";"
      ;

SvgInEvent : Name
           | "if" Expression ":" Name OptSvgInEventElifs "else" Name "end"
           | "if" Expression ":" Name SvgInEventElifs "end"
           ;

OptSvgInEventElifs : /* empty */
                   | SvgInEventElifs
                   ;

SvgInEventElifs : "elif" Expression ":" Name
                | SvgInEventElifs "elif" Expression ":" Name
                ;

PrintFile : "printfile" StringToken ";"
          ;

Print : "print" PrintTxt OptPrintFors OptPrintWhen OptPrintFile ";"
      ;

PrintTxt : Expression
         | "pre" Expression
         | "post" Expression
         | "pre" Expression "post" Expression
         ;

OptPrintFors : /* empty */
             | "for" PrintFors
```

```
            ;

PrintFors : PrintFor
          | PrintFors "," PrintFor
          ;

PrintFor : "event"
         | "time"
         | Name
         | "initial"
         | "final"
         ;

OptPrintWhen : /* empty */
             | "when" Expression
             | "when" "pre" Expression
             | "when" "post" Expression
             | "when" "pre" Expression "post" Expression
             ;

OptPrintFile : /* empty */
             | "file" StringToken
             ;

////////////////////////////////////////////////////////////////////////////

Types : Type
      | Types "," Type
      ;

EventType : "void"
          | Type
          ;

Type : "bool"
     | "int"
     | "int" "[" Expression ".." Expression "]"
     | "real"
     | "string"
     | "list" Type
     | "list" "[" Expression "]" Type
     | "list" "[" Expression ".." Expression "]" Type
     | "set" Type
     | "dict" "(" Type ":" Type ")"
     | "tuple" "(" Fields ")"
     | "func" Type "(" ")"
     | "func" Type "(" Types ")"
     | "dist" Type
     | Name
     ;
```

```
Fields : Field
       | Fields ";" Field
       ;

Field : Type Identifiers
      ;

////////////////////////////////////////////////////////////////////////////

Expressions : Expression
            | Expressions "," Expression
            ;

OptExpression : /* empty */
              | Expression
              ;

Expression : OrExpression
           | OrExpression "=>" OrExpression
           | OrExpression "<=>" OrExpression
           ;

OrExpression : AndExpression
             | OrExpression "or" AndExpression
             ;

AndExpression : CompareExpression
              | AndExpression "and" CompareExpression
              ;

CompareExpression : AddExpression
                  | CompareExpression "<"  AddExpression
                  | CompareExpression "<=" AddExpression
                  | CompareExpression "="  AddExpression
                  | CompareExpression "!=" AddExpression
                  | CompareExpression ">=" AddExpression
                  | CompareExpression ">"  AddExpression
                  | CompareExpression "in" AddExpression
                  | CompareExpression "sub" AddExpression
                  ;

AddExpression : MulExpression
              | AddExpression "+" MulExpression
              | AddExpression "-" MulExpression
              ;

MulExpression : UnaryExpression
              | MulExpression "*" UnaryExpression
              | MulExpression "/" UnaryExpression
              | MulExpression "div" UnaryExpression
              | MulExpression "mod" UnaryExpression
```

```
              ;

UnaryExpression : FuncExpression
                | "-" UnaryExpression
                | "+" UnaryExpression
                | "not" UnaryExpression
                | "sample" FuncExpression
                ;

FuncExpression : ExpressionFactor
               | FuncExpression "[" Expression "]"
               | FuncExpression
                 "[" OptExpression ":" OptExpression "]"
               | FuncExpression "(" ")"
               | FuncExpression "(" Expressions ")"
               | StdLibFunction "(" ")"
               | StdLibFunction "(" Expressions ")"
               ;

ExpressionFactor : "true"
                 | "false"
                 | NUMBERTK
                 | REALTK
                 | StringToken
                 | "time"
                 | "[" "]"
                 | "[" Expressions "]"
                 | "[" Expression "for" ExprLocalAddrs "in" Expression "]"
                 | "[" Expression "for" ExprLocalAddrs "in" Expression "if" Expression
"]"
                 | "{" "}"
                 | NonEmptySetExpression
                 | "{" DictPairs "}"
                 | "(" Expression "," Expressions ")"
                 | "<" Type ">" ExpressionFactor
                 | "if" Expressions ":" Expression
                   OptElifExprs "else" Expression "end"
                 | "switch" Expression ":" SwitchBody "end"
                 | "(" Expression ")"
                 | Name
                 | Name "'"
                 | "?"
                 | "self"
                 ;

ExprLocalAddrs : ExprLocalAddr
               | ExprLocalAddrs "," ExprLocalAddr
               ;

ExprLocalAddr : Identifier
              | "(" ExprLocalAddr "," ExprLocalAddrs ")"
```

```
             ;

NonEmptySetExpression : "{" Expressions "}"
                      ;

DictPairs : Expression ":" Expression
          | DictPairs "," Expression ":" Expression
          ;

OptElifExprs : /* empty */
             | OptElifExprs "elif" Expressions ":" Expression
             ;

SwitchBody : SwitchCases
           | SwitchCases "else" Expression
           |             "else" Expression
           ;

SwitchCases : "case" Expression ":" Expression
            | SwitchCases "case" Expression ":" Expression
            ;

Name : Identifier
     | RELATIVENAMETK
     | ABSOLUTENAMETK
     | ROOTNAMETK
     ;

////////////////////////////////////////////////////////////////////////////

OptDoubleAnnos : /* empty */
               | OptDoubleAnnos DoubleAnnotation
               ;

DoubleAnnotation : DOUBLE_ANNOTATION_NAMETK
                 | DOUBLE_ANNOTATION_NAMETK "(" ")"
                 | DOUBLE_ANNOTATION_NAMETK "(" AnnotationArgs OptComma ")"
                 ;

OptAnnos : /* empty */
         | OptAnnos Annotation
         ;

Annos : Annotation
      | Annos Annotation
      ;

Annotation : REGULAR_ANNOTATION_NAMETK
           | REGULAR_ANNOTATION_NAMETK "(" ")"
           | REGULAR_ANNOTATION_NAMETK "(" AnnotationArgs OptComma ")"
           ;
```

```
AnnotationArgs : AnnotationArg
               | AnnotationArgs "," AnnotationArg
               ;

AnnotationArg : IDENTIFIERTK ":" Expression
              | RELATIVENAMETK ":" Expression
              | Expression
              ;

OptComma : /* empty */
         | ","
         ;

SupKind : "plant"
        | "requirement"
        | "supervisor"
        ;

StdLibFunction : "acosh"
               | "acos"
               | "asinh"
               | "asin"
               | "atanh"
               | "atan"
               | "cosh"
               | "cos"
               | "sinh"
               | "sin"
               | "tanh"
               | "tan"
               | "abs"
               | "cbrt"
               | "ceil"
               | "del"
               | "empty"
               | "exp"
               | "floor"
               | "fmt"
               | "ln"
               | "log"
               | "max"
               | "min"
               | "pop"
               | "pow"
               | "round"
               | "scale"
               | "sign"
               | "size"
               | "sqrt"
               | "bernoulli"
```

```
                | "beta"
                | "binomial"
                | "constant"
                | "erlang"
                | "exponential"
                | "gamma"
                | "geometric"
                | "lognormal"
                | "normal"
                | "poisson"
                | "random"
                | "triangle"
                | "uniform"
                | "weibull"
                ;
```

3.1.3. CIF XML files

While CIF has a textual syntax (see the lexical syntax and grammar), in certain cases it may be more convenient to work with CIF specifications as XML files. For instance:

- When debugging tools that produce CIF models, the XML file format contains all internal details, such as the types of expressions.
- For processing of CIF specifications by other tools, having an XML syntax can be useful as well. Those tools do not have to parse and type check CIF specifications by themselves, as the XML files contain the type-checked specifications.

All CIF tools bundled with Eclipse ESCET support writing CIF specifications as XML files. To write a CIF XML file instead of a regular CIF file in the CIF textual syntax, use the `.cifx` file extension for the output CIF file.

All CIF tools bundled with Eclipse ESCET also support reading CIF specifications from XML files. If an input CIF specification has the `.cifx` file extension, it is read as an XML file.

The CIF XML files are a special type of XML, namely XMI. XMI stands for 'XML Metadata Interchange' and is an OMG standard.

XML files produced by CIF tools should be considered to be version-specific. Different versions of CIF may read and write XML files that are not compatible with previous versions. The CIF release notes do not indicate such incompatibilities.

For more information on how to obtain and work with CIF XML files, check out the relevant section of the developer documentation.

3.2. SVG visualization and interaction

3.2.1. CIF/SVG declarations

CIF supports visualizing the state of a model using SVG images, as well interacting with these images during for instance a simulation. To better understand these concepts, see the CIF tutorial lessons on SVG visualization and SVG interaction. Here, we focus more on their technical details.

SVG images can be connected to the CIF model, the images can be updated based on the state of the model, and clicks on elements of the images can be used to influence the behavior of the model. All of these connections are specified in a CIF model using CIF/SVG declarations (sometimes also called SVG declarations, for simplicity).

The following types of CIF/SVG declarations are available:

- SVG file declarations: Specifies which SVG image file to use.
- CIF/SVG copy declarations: Specifies that a part of an SVG image should be copied.
- CIF/SVG move declarations: Specifies that a part of an SVG image should be moved.
- CIF/SVG output mappings: Specifies how to update a certain property of an element of an SVG image.
- CIF/SVG input mappings: Specifies how interaction with an SVG image, by clicking on certain elements of an image, affects the behavior of the model.

SVG file declarations

Before any other CIF/SVG declarations can be specified, you need to indicate to which SVG image (file) they apply, using an SVG file declaration. A typical SVG file declaration looks like this:

```
svgfile "some_file.svg";
```

The SVG files are declared based on their file names. If possible, it is recommended to use the same file name for the CIF specification and for the SVG image, only with different file extensions. This way, it is easy to see which images goes with which CIF specification.

Supported SVG images

CIF supports version 1.1 of the SVG standard. Features from version 2 of the SVG standard are not supported. For a few SVG version 2 features, the CIF type checker gives a warning if such a feature is used in an SVG image that is used in a CIF model.

Paths

It is possible to use relative (or absolute) paths for the SVG files. For instance:

```
svgfile "svg_files/some_file.svg";
svgfile "../images/some_file.svg";
```

For the first SVG file declaration, the tools will look for an `svg_files` directory in the same directory as the CIF file, and for a `some_file.svg` file in that `svg_files` directory. For the second SVG file declaration, the tools will look for an `images` directory in the parent directory of the directory that contains the CIF file, and for a `some_file.svg` file in that `images` directory.

Files and directories are separated using slashes (`/`), regardless of the used operating system. It is also allowed to use backslashes (`\`), which are more common on Microsoft Windows operating systems, but they need to be escaped as `\\` for this to work, making it easier to use slashes (`/`) instead.

Scoping

CIF/SVG file declarations (and all 'other' CIF/SVG declarations) can be specified in all components. That is, they can be specified in the top level scope of the specification, in groups and group definitions, as well as in automata and automaton definitions. An SVG file declaration that is specified in a certain scope applies to that scope, as well as all child scopes that don't have an SVG file declaration of their own. Additionally, all 'other' CIF/SVG declarations (such as SVG output mappings) may include a local SVG file declaration, that only applies to that specific CIF/SVG declaration.

For instance, consider the following CIF specification:

```
svgfile "file1.svg";

svgout id "a" attr "b" value 5;

svgout id "c" attr "d" value 6 file "file2.svg";

group g:
  svgfile "file3.svg";

  svgout id "e" attr "f" value 7;

  svgout id "g" attr "h" value 8;

  svgout id "i" attr "j" value 9 file "file4.svg";

  group h:
    svgfile "file5.svg";
  end

  group i:
    svgout id "k" attr "l" value 10;
  end
end
```

The mapping for the SVG element with id `a` is declared in the top level scope of the specification,

and does not have a local SVG file declaration. As such, the SVG file declaration from the top level scope (`file1.svg`) applies to it. The mapping for the SVG element with id `b` is declared in the same scope, but has a local SVG file declaration (`file2.svg`). The local SVG file declaration (`file2.svg`) overrides the SVG file declaration from the top level scope (`file1.svg`).

Group `g` also has an SVG file declaration (`file3.svg`). This overrides the SVG file declaration from the top level scope (`file1.svg`). The SVG file declaration from group `g` (`file3.svg`) thus applies to the mappings for ids `e` and `g`, as they are declared in the same scope, and don't have local SVG file declarations. The mapping for SVG element id `i` has a local SVG file declaration (`file4.svg`), and as such the SVG file declaration from group `g` (`file3.svg`) does not apply to it.

The SVG file declaration (`file5.svg`) from group `g.h` does not apply to any mappings. The image is connected to the model, but its is not updated based on the state of the model.

Group `g.i` does not have an SVG file declaration, so the SVG file declaration (`file3.svg`) from group `g` (the parent scope of group `g.i`) applies to the mapping for SVG element id `k`.

See also the Push SVG file declarations into other CIF/SVG declarations CIF to CIF transformation.

Completeness and uniqueness

Every CIF/SVG declaration that is not itself an SVG file declaration, requires an SVG file declaration, in its scope, in one of its ancestor scopes, or locally in that CIF/SVG declaration itself.

In each scope, there may be at most one SVG file declaration.

In every scope (or component), in principle at most one SVG file declaration may be specified. The exception is that for a single scope, specified in multiple CIF files (when imports are used), if the SVG file declarations refer to the same image file, they are merged. If however the two CIF files refer to different image files, this is still considered an error. It is also considered an error if two SVG file declarations are present in a single scope, in a single file, regardless of whether they refer to the same image file or not. In the end, after processing imports, for every scope there must be a unique SVG file declaration, if one is declared at all.

If the same SVG image file is specified multiple times, in different SVG file declarations, the corresponding image is connected to the model only once. For instance, consider the following CIF specification:

```
svgout id "..." attr "..." value 5 file "some_file.svg";

svgout id "..." attr "..." value 6 file "some_file.svg";
```

Even though both output mappings specify an SVG file locally, they use the same file. The SVG image will thus be connected to the model only once, and both mappings apply to the same image.

CIF/SVG copy declarations

A CIF/SVG copy declaration can be used to create copies of existing graphical elements of an SVG image. This page explains the technical details of such copy declarations. For concrete examples, see the lamps example or the buffers/products example.

Example

Consider the following SVG file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <rect style="fill:red;"   id="rect"  width="100" height="100" x="20" y="20"/>
  <g id="grp">
    <rect style="fill:green;" id="rect2" width="100" height="100" x="50" y="50"/>
  </g>
  <rect style="fill:blue;"  id="rect3" width="100" height="100" x="80" y="80"/>
</svg>
```

It features three rectangles, with ids `rect`, `rect2`, and `rect3`, each with the same size. They have different `fill` colors and positions. The second rectangle is enclosed in a group with id `grp`. Also consider the following CIF file:

```
svgfile "svgcopy.svg";

svgcopy id "rect" pre "a";
svgcopy id "grp" post "b";
svgcopy id "rect3" pre "_" post "x";
```

The effect of the copy declarations on the SVG image is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <rect style="fill:red;"   id="arect"   width="100" height="100" x="20" y="20"/>
  <rect style="fill:red;"   id="rect"    width="100" height="100" x="20" y="20"/>
  <g id="grpb">
    <rect style="fill:green;" id="rect2b" width="100" height="100" x="50" y="50"/>
  </g>
  <g id="grp">
    <rect style="fill:green;" id="rect2" width="100" height="100" x="50" y="50"/>
  </g>
  <rect style="fill:blue;"  id="_rect3x" width="100" height="100" x="80" y="80"/>
```

```
  <rect style="fill:blue;"  id="rect3"   width="100" height="100" x="80" y="80"/>
</svg>
```

Syntax

The general syntax of CIF/SVG copy declarations is:

```
svgcopy id <orig-elem-id> pre <prefix>;

svgcopy id <orig-elem-id> post <postfix>;

svgcopy id <orig-elem-id> pre <prefix> post <postfix>;

svgcopy id <orig-elem-id> pre <prefix> file "...";
```

The `svgcopy` keyword is followed by a specification of an SVG element id. Every copy declaration must specify the id of an SVG element, and an SVG element with that id must exist in the original SVG image, or must exist after applying one of the other copy declarations of the same CIF specification. The `id` indicates the element that is to be copied.

The prefix and postfix are both optional, but at least one of them must be specified. The prefix and postfix are used to generate unique ids for the copied elements. The prefix (if any) is prepended to the ids of the copied elements, while the postfix (if any) is appended to the ids of the copied elements. Prefixes (if specified) must be valid SVG name prefixes, which means they must be valid SVG names, just as SVG element ids. Similarly, postfixes (if specified) must be valid SVG name postfixes, which means they must be valid SVG names, but may start with dashes (`-`), dots (`.`), and numbers (`0` to `9`). Similar to ids, expressions may be used for both prefixes and postfixes, as long as they result in a `string` typed value when they're evaluated.

If we look at the rectangles example above, we see that the `rect` element is copied, and that the `id` of the copied element is prefixed with `a`, resulting in `arect` as `id` of the copy. Similarly, `rect3` is prefixed with `_` and postfixed with `x` resulting in `_rect3x` as `id` of the copy.

The `grp` element is copied as well, with a `b` postfix. This results in a copied group with `grpb` as its `id`. Group `grp` also contains the rectangle with id `rect2`. When an element is copied, all its contents are copied as well. That is, the copy is recursive. This means that the `rect2` element is copied as well, and that copy is also given a `b` postfix, resulting in `rect2b` as its `id`.

In the syntax examples above, we see that it is also allowed to declare a local SVG file declaration, that only applies to that specific copy declaration.

Placement

The copies are added to the SVG image, as siblings of their originals. That is, the copies are added just before their originals. In the rectangles example above, this means that the `arect` copy is added

just before the `rect` original, and the `grpb` copy is added just before the `grp` original.

The order of the `rect`, `rect2`, and `rect3` rectangles is important, in that it determines their *rendering order*. Elements that are listed first, get *rendered* or *painted* first. Subsequent elements are painted on top of previously painted elements. Thus, the red `rect` rectangle is painted first, and is partially covered by the green `rect2` rectangle, which is painted second and is partially covered by the blue `rect3` rectangle, which is painted third.

By including copies near their originals (as siblings), the copies get a similar rendering order.

It is not allowed to copy the root element (the element with id `root` in the rectangles example above), as that would result in two root elements, which is not allowed by the SVG standard.

Uniqueness

All copied elements (the element that is copied, its children, the children of its children, etc) have their ids pre- and postfixed to create their new ids. These ids must be unique in the SVG image. That is, the new ids must not yet exist in the original SVG image, or have been used for previously copied elements. The unique ids are required to ensure that we can still uniquely identify all elements by their ids, and we can thus use the ids to uniquely indicate to which elements other CIF/SVG declarations apply.

Consider once again the rectangles example above:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <rect style="fill:red;"   id="rect"  width="100" height="100" x="20" y="20"/>
  <g id="grp">
    <rect style="fill:green;" id="rect2" width="100" height="100" x="50" y="50"/>
  </g>
  <rect style="fill:blue;"  id="rect3" width="100" height="100" x="80" y="80"/>
</svg>
```

We add the following copy declaration:

```
svgcopy id "rect" post "2";
```

This would result in the `rect` element being copied, and the copy being given a `rect2` id. However, there is already an element with that id. As such, the copy declaration is illegal.

At least a prefix or a postfix is required for every copy declaration, as otherwise the new ids would be identical to the original ids.

Overlap

Consider the following SVG file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <g id="g">
    <rect style="fill:green;" id="r" width="100" height="100" x="50" y="50"/>
  </g>
</svg>
```

We use the following copy declarations:

```
svgcopy id "g" post "a";
svgcopy id "r" post "b";
```

Both the `g` element and the `r` element are copied. However, the `r` element is a part of the `g` element. Therefore, the `r` element is copied twice, once for the copy of that element itself, and once for the copy of its parent (`g`). This leads to the following SVG image:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <g id="ga">
    <rect style="fill:green;" id="ra" width="100" height="100" x="50" y="50"/>
  </g>
  <g id="g">
    <rect style="fill:green;" id="rb" width="100" height="100" x="50" y="50"/>
    <rect style="fill:green;" id="r" width="100" height="100" x="50" y="50"/>
  </g>
</svg>
```

However, if we switch the order of the copies, we get:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <g id="ga">
    <rect style="fill:green;" id="rba" width="100" height="100" x="50" y="50"/>
    <rect style="fill:green;" id="ra" width="100" height="100" x="50" y="50"/>
  </g>
  <g id="g">
    <rect style="fill:green;" id="rb" width="100" height="100" x="50" y="50"/>
```

```
    <rect style="fill:green;" id="r" width="100" height="100" x="50" y="50"/>
  </g>
</svg>
```

Copying an element twice by copying the element itself (e.g. `g`), but also copying an element that it contains (e.g. `r`) is not recommended. As seen above, the outcome depends on the order, and can be very confusing. CIF gives a warning in case such overlapping copies are used. Generally, instead of copying the element twice, what is really wanted, is to copy the copied element. This can be done as follows:

```
svgcopy id "g" post "a";
svgcopy id "ra" post "b";
```

And this results in the following SVG file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="250" height="250" id="root" version="1.1">
  <g id="ga">
    <rect style="fill:green;" id="rab" width="100" height="100" x="50" y="50"/>
    <rect style="fill:green;" id="ra" width="100" height="100" x="50" y="50"/>
  </g>
  <g id="g">
    <rect style="fill:green;" id="r" width="100" height="100" x="50" y="50"/>
  </g>
</svg>
```

See the buffers/products example for a more concrete example of the usefulness of copying copied elements.

Not that overlap is only detected for copying an element and an element that it contains. Copying the same element multiple times is perfectly valid and useful, and does not result in warnings.

Application

Copy declarations are only applied once, before any CIF/SVG move declarations and CIF/SVG output mappings.

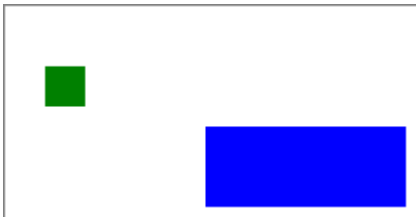
CIF/SVG move declarations

A CIF/SVG move declaration can be used to move parts of an SVG image to a certain position, regardless of transformations applied to the elements that are moved, or any surrounding groups and layers, etc. This page explains the technical details of such move declarations. For a concrete example, see the lamps example.

Example

Consider the following SVG file:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="400" height="400" id="root" version="1.1">
  <g id="grp1">
    <rect style="fill:green;" id="rect1" width="20" height="20" x="5" y="10"
          transform="translate(15, 20)"/>
  </g>
  <g id="grp2" transform="scale(5, 2)">
    <rect style="fill:blue;" id="rect2" width="20" height="20" x="5" y="10"
          transform="translate(15, 20)"/>
  </g>
</svg>
```



It features a group `grp1`, with a green rectangle `rect1`. It is 20 by 20 pixels in size. Its position is 20 pixels from the left (`x` is 5, but the horizontal translation is 15 pixels) and 30 pixels from the top (`y` is 10, but the vertical translation is 20 pixels).

Group `grp2` and rectangle `rect2` are similar. Rectangle `rect2` is blue, and group `grp2` is scaled 5 times horizontally, and 2 times vertically. This means that rectangle `rect2` is 100 by 40 pixels in size (5 times 20, and 2 times 20). It also means that the position of `rect2` is 100 pixels from the left (5 times 20) and 60 pixels from the top (2 times 30).

Also consider the following CIF file:

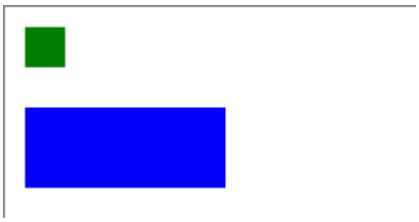
```
svgfile "svgmove.svg";

svgmove id "rect1" to 10, 10;
svgmove id "rect2" to 10, 50;
```

The upper left corner of the bounding box of rectangle `rect1` is moved to 10 pixels from the left side, and 10 pixels from the top. The upper left corner of the bounding box of rectangle `rect2` is moved to 10 pixels from the left side, and 50 pixels from the top. The effect of the move declarations on the SVG image is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

```
<svg xmlns="http://www.w3.org/2000/svg"
     xmlns:svg="http://www.w3.org/2000/svg"
     width="400" height="400" id="root" version="1.1">
  <g id="grp1">
    <rect style="fill:green;" id="rect1" width="20" height="20" x="5" y="10"
          transform="translate(15, 20) translate(-10.000000,-20.000000)"/>
  </g>
  <g id="grp2" transform="scale(5, 2)">
    <rect style="fill:blue;" id="rect2" width="20" height="20" x="5" y="10"
          transform="translate(15, 20) translate(-18.000000,-5.000000)"/>
  </g>
</svg>
```



Note the translations that have been added to the `transform` attributes of the rectangles. The additional translation for rectangle `rect1` is -10 in the x direction and -20 in the y direction, as the rectangle has been moved from position (20, 30) to position (10, 10).

Rectangle `rect2` has been moved from position (100, 60) to position (10, 50). You might have expected a horizontal translation of -90 and a vertical translation of -10. However, its added translation is for -18 pixels horizontally, and -5 pixels vertically. This is because the scaling applies to this translation as well, and 5 * 18 = 90, and 2 * 5 = 10.

The scaling of the surrounding group complicates the calculations for the transformations of its children (`rect2` in this case). Moving elements to certain positions becomes more and more complicated when the elements themselves have transformations, their parents have transformations, the parents of their parents have transformations, etc. This especially holds for non-translation transformations, such as `scale`, `rotate`, etc. The CIF/SVG move declarations make moving easy, by allowing you to specify the target position, regardless of the existence of such transformations.

Syntax

The general syntax of CIF/SVG move declarations is:

```
svgmove id <id> to <x>, <y>;

svgmove id <id> to <x>, <y> file "...";
```

The `svgmove` keyword is followed by a specification of an SVG element id. Every move declaration must specify the id of an SVG element, and an SVG element with that id must exist in the SVG image. The `id` indicates the element that is to be moved.

The x and y target coordinates are both mandatory, and indicate the coordinates of the upper left corner of the bounding box of element that is being moved, relative to the upper left corner of the canvas, after the move is completed. Similar to ids, expressions may be used for both the x and y coordinates, as long as they result in a `int` or `real` typed value when they're evaluated. That is, the x and y coordinates must be numeric.

In the syntax examples above, we see that it is also allowed to declare a local SVG file declaration, that only applies to that specific move declaration.

Moving graphical elements

In order for an element to be moved, it must have a graphical representation. Elements that have a graphical representation include rectangles, paths, text labels, groups, etc. However, certain elements don't have a graphical representation, and thus can not be moved. An example of elements that can not be moved are the `tspan` sub-elements of `text` elements. The `tspan` sub-elements don't have graphical representations of their own, they are part of the text labels, and only the `text` elements have graphical representations.

Conflicting transformations

The SVG standard allows for powerful transformations, including the `scale` and `matrix` transformations. Now assume we use a `scale(0, 0)` transformation. Then all coordinates are multiplied by zero, always resulting in zero. As such, the entire image is scaled into zero by zero pixels. In such cases, there is no translation that can be added to an element, that after application of the scaling transformation, moves it to a certain position. Thus, if such conflicting transformations are present, the move can not be accomplished, and the CIF/SVG move declaration is considered illegal. It should be noted however, that such conflicting transformations are rare, and usually indicate an error has been made in those transformations.

Application

Move declarations are only applied once, after any CIF/SVG move declarations and before any CIF/SVG output mappings.

CIF/SVG output mappings

For each update of a single attribute of an element in the SVG image, an *output mapping* is needed. An output mapping essentially defines a connection between the CIF specification and the SVG image, for updating that specific attribute of that specific element.

A simple output mapping

Assume we have the following CIF specification:

```
cont pos der 1.0;
```

Where variable `pos` is the position of some object, say a box. Then the following output mapping:

```
svgout id "box" attr "x" value pos;
```

Keeps the value of the `x` attribute of the SVG element with id `box` equal to the value of continuous variable `pos`.

Syntax

The general syntax of CIF/SVG output mappings is:

```
svgout id <id> attr <attr> value <value>;
svgout id <id> attr <attr> value <value> file <file>;
svgout id <id> text        value <value>;
svgout id <id> text        value <value> file <file>;
```

The `svgout` keyword is followed by a specification of an SVG element id. Every output mapping must specify the id of an SVG element, and an SVG element with that id must exist in the SVG image. The `id` indicates the element for which an attribute or text is to be updated.

The `id` is followed by a specification of what part of the SVG element is to updated. That is, either an attribute or text (label) is updated.

After that, the value of the attribute or text (label) is specified.

In the syntax examples above, we see that it is also allowed to declare an optional local SVG file declaration, that only applies to that specific output declaration.

SVG element id

Every output mapping needs to specify the id of the SVG element to update. The id is checked, to make sure an SVG element with that id actually exists in the SVG image.

Only valid SVG names may be used as ids. Valid SVG names start with a letter (`a` to `z`, `A` to `Z`), an underscore (`_`), or a colon (`:`), and may be followed by more of the same characters, as well as dashes (`-`), dots (`.`), and numbers (`0` to `9`).

Ids must be put between double quotes. For instance, to use `some-name` as id, it must be written as `"some-name"`.

Instead of using a hard coded SVG element id, it is also allowed to use an expression that when evaluated results in the SVG element id. Such an expression must evaluate to a `string` typed value,

and must be statically evaluable (must for example not refer to variables that can change based on the model's state). For instance:

```
// Hard coded SVG element id.
svgout id "box" attr "width" value ...;

// Computed SVG element id.
const string c = "bo";
svgout id c + "x" attr "width" value ...;
```

The first output mapping applies to the SVG element with the id `box`. The `box` id is hard coded into the output mapping. The second output mapping uses the expression `c + "x"` to specify the SVG element id. Once the expression is evaluated (its value is computed), this results in the `string` typed value `"box"`. As such, the second mapping applies to the SVG element with id `box` as well.

The expressions may also refer to for instance algebraic parameters of automaton definitions. Since different values can be given to the parameters for each instantiation, this allows the SVG element id to vary for the different instantiations of the automaton definition. This particular usage allows for reuse and scalable solutions. The lamps example and workstation example are good examples of this usage. This usage is often combined with CIF/SVG copy declarations and CIF/SVG move declarations.

SVG attribute name

Every output mapping needs to specify the name of the attribute of the SVG element to update (or `text`, see the Text labels section). Similarly to specifying ids, attribute names must be valid SVG names. They also need to be put between double quotes. Unlike SVG element ids, it is not possible to use expressions to specify attribute names. That is, only hard coded names can be used to specify attribute names.

It is not allowed to change the ids of SVG elements, as it could lead to duplicate or missing ids. As such, output mappings for the `id` attribute (in any casing) are explicitly not supported. Similarly, changing the `style` attribute (in any casing) using an output mapping is explicitly not supported, to avoid conflicting style changes. See also the SVG presentation attributes vs CSS style attributes section for more information about the `style` attribute.

The supported attributes for an SVG element depend on what kind of element it is. For instance, a `rect` element can have an `x` and `y` position, a `width` and `height`, a `fill` color, etc. A `g` element (a group) however, does not support those position attributes. It is beyond the scope of this document to go into more detail. The CIF type checker however, will warn about using attribute names that are not supported according to the SVG standard (to catch simple spelling mistakes), and specifying attributes on elements that don't support them. Consult the official SVG specification for further details. Also, the CIF tutorial SVG visualization examples showcase several commonly used attributes.

SVG presentation attributes vs CSS style attributes

SVG uses two kinds of attributes: SVG presentation attributes, and CSS style attributes. For instance, in the following partial SVG image:

```
<rect fill="red"        id="rect1" width="25" height="25" x="65" y="5"/>
<rect style="fill:red;" id="rect2" width="25" height="25" x="15" y="5"/>
```

Rectangle `rect1` uses the SVG presentation attribute `fill` to specify the fill color the rectangle. Rectangle `rect2` uses the CSS style attribute `fill` to specify the fill color the rectangle. The SVG presentation attributes are all direct attributes of the SVG element. CSS style attributes on the other hand, are always part of the SVG presentation attribute `style`.

CSS stands for *Cascading Style Sheets*, and it is a style sheet language used for describing the look and formatting of a document. It's best known for its use in styling web pages written in HTML. Like SVG, CSS is an international standard developed and maintained by the World Wide Web Consortium (W3C). SVG reuses the CSS standard for its CSS style attributes. The most commonly used attributes for certain elements are also available in the SVG standard as SVG presentation attributes. Furthermore, SVG adds additional presentation attributes not found in CSS.

If an attribute is specified on an element using both an SVG presentation attribute, and a CSS style attribute, the CSS style attribute usually takes precedence. The output mappings handle all of this automatically and transparently. That is, you just specify the attribute name in the header of the output mapping, and automatically the proper attribute is updated, regardless of whether it is an SVG presentation attribute, or a CSS style attribute. It is explicitly not supported to change the `style` attribute directly (using an output mapping), in order to avoid conflicting style changes when other output mappings update CSS style attributes that are part of that same `style` SVG presentation attribute.

Text labels

Changing the text of text labels is a special case, as text is not stored in attributes. Consider the following partial SVG image, which contains a text label (created using Inkscape, and with some of the irrelevant details omitted):

```
<text x="5" y="5" id="text1" style="font-size:16px;">
  <tspan x="5" y="5" id="tspan2">
    The actual text.
  </tspan>
</text>
```

Here we see a `text` element with id `text1`, which contains a `tspan` element with id `tspan2`. The `tspan` element contains some text. To change the text of this text label, use one of the following mappings:

```
svgout id "text1" text value ...;
```

```
svgout id "tspan2" text value ...;
```

Here, instead of the `attr` keyword followed by the name of the attribute, we simply specify the `text` keyword. We can either use the id of the `text` element, or the id of the `tspan` element. More precisely, we can use the id of an element that contains text (such as the `tspan` element in our example), or an element (such as the `text` element in our example) that has a single child, which contains text, or an element that has a single child, which has an single child, which contains text, etc.

In practice, it is usually the easiest to use the *Object Properties* panel of Inkscape, to get the properties of the text label. The *ID* that is then shown, is the id of the `text` element.

Output mapping value

In order for an output mapping to be able to update attributes or text labels, it needs a value. The simple output mapping example already showed how a value can be specified. This section further explains the details of the specification of such values.

The `value` keyword is followed by an expression. This expression may for instance be a literal value, such as `1`, `true`, `1.0`, `"some text"`, or `[1, 2]`. At first, it might not seem useful to use a literal as a value for an output mapping, since the mapping will then produce the same value every time it is applied. However, the rate example shows why this can in fact be very useful.

The expressions however, may also contain references to variables (discrete, input, continuous, or algebraic), functions, constants, locations, etc. For instance, we could use the value of a continuous variable and multiply it by two, by using `x * 2` as expression, assuming that the continuous variable is named `x`. Since we may refer to the state of the CIF specification, the value of the expression can change at runtime, resulting in different values being mapped to the image.

The expressions can also include `if` expressions and switch expressions, which are particularly powerful in this context. Using these expressions, a different value can be used for several different conditions. See the sun/moon example and the workstation example for an example of using these expressions in an output mapping.

Quoting and escaping

The value expression of an output mapping may be of any type. The result of evaluating the expression is converted to a textual representation that closely resembles the textual syntax of CIF, before using it to set the value of an attribute, or the text of a text label. For string literals this means that the text is escaped, and double quotes are added. If however the result of the expression is a `string` typed value, then that string is used 'as is' (without quoting, and without escaping). Thus, consider the following examples:

```
svgout id "..." attr "..." value ["a\"b"];
svgout id "..." attr "..." value "a\"b";
```

The value of the first output mapping is a list that contains a single string value, while the value of the second output mapping is a string value directly. This results in the following texts being used as values of the attributes:

```
["a\"b"]
a"b
```

That is, in general string values are quoted and escaped, as is the case for the first example, where the result is a list. However, if the entire result is a string, as is the case with the second example, the string value is used 'as is', without quoting and without escaping.

Whitespace handling

If the textual result of an output mapping contains a new line character (for instance due to including `\n` in a string literal or format pattern), the new line character is ignored by SVG. Furthermore, tabs (`\t`) are replaced by a single space character. Finally, by default consecutive spaces (including tabs) are merged together into a single space, and spaces at the beginning and end of the output of a mapping are discarded by SVG.

This essentially renders `\n` and `\t` useless, as the first is ignored, and for the second spaces can be used instead.

This also means that it is not possible to set multiple lines of text using a single output mapping. This is a fundamental restriction of SVG, not of CIF/SVG output mappings. In order to set multiple lines of text, multiple output mappings are needed, where each mapping sets a different text label (a different line).

Note that if you enter multiple lines of text in Inkscape, this results in a single `text` element, with multiple `tspan` child elements. The actual text of such `tspan` elements can then be updated using an output mapping for each `tspan` element. The benefit over multiple text labels (multiple `text` elements with a single `tspan` child element each) is that the one `text` element can be moved, making all the `tspan` child elements automatically move with it. That is, the lines of text are kept together by Inkscape.

Uniqueness of output mappings

All output mappings must be unique, per SVG image. That is, no two mappings may use the exact same SVG element id and attribute name. Similarly, no two mappings may update the same text. That is, for the example above, it is not allowed to specify two mappings for the text of the SVG element with id `text1`. However, since both elements `text1` and `tspan1` have the same text, it is also not allowed to specify output mappings for the text of those two elements.

Application

SVG output mappings are only applied after any CIF/SVG copy declarations and CIF/SVG move declarations have already been applied. The output mappings are applied for the initial state, as well as subsequent states that are reached.

CIF/SVG input mappings

CIF/SVG input mappings allow for interaction with SVG images. The input mappings allow for interaction, by clicking on elements of an SVG image, which then affect the behavior of the specification. See the CIF tutorial lesson on SVG interaction to better understand this interactivity.

Concretely, a CIF/SVG input mapping is a CIF/SVG declaration that couples an id of an SVG element to an event or one or more updates in the CIF specification.

Interactive elements

Every input mapping must specify the id of an SVG element, and an SVG element with that id must exist in the SVG image. Specifying an input mapping for an SVG element turns that SVG element into an *interactive SVG element*. The SVG input mapping then determines what happens when such an interactive element is clicked. There are two types of interactions: enabling an event and executing some updates.

Note that it is also possible to make a group of SVG elements (an `svg:g` element) an interactive element. By doing so, all the elements that are part of the group together become one single interactive element.

One practical example where this is very useful, is for buttons with a text label. In SVG, this requires the use of a rectangle and a text label. By grouping the rectangle and text label together in an SVG group, and adding an input mapping to that group, the rectangle and label can both be clicked. Clicking on either of them results in the corresponding input mapping being applied.

It is strongly recommended not to make a child (or descendant in general) of an interactive SVG element interactive as well.

Single event input mapping

The simplest form of an SVG input mapping maps to a single event. It has the following syntax:

```
svgin id <id> event <event>;
svgin id <id> event <event> file <file>;
```

The `svgin` keyword is followed by a specification of an SVG element id.

The `id` is followed by a reference to an event in the specification. This event is disabled until the interactive element, the one with the given id, is clicked. Clicking the interactive element enables the event such that at most one transition may be taken for it. After taking the transition, the event is again disabled until the element is clicked once more. If after clicking the element the event is not

enabled in the current state of the model, for instance because an automaton in the model can't synchronize on it, the click is ignored.

It is also allowed to declare an optional local SVG file declaration, that only applies to that specific input declaration.

`if` event input mapping

An *if event input mapping* is similar to a single event input mapping, but allows to map to different events depending on the state of the model. Its syntax is as follows:

```
svgin id <id> event if <expr>: <event> else <event> end;
svgin id <id> event if <expr>: <event> elif <expr>: <event> ... else <event> end;
svgin id <id> event if <expr>: <event> elif <expr>: <event> ... end;

svgin id <id> event if <expr>: <event> else <event> end file <file>;
svgin id <id> event if <expr>: <event> elif <expr>: <event> ... else <event> end file
<file>;
svgin id <id> event if <expr>: <event> elif <expr>: <event> ... end file <file>;
```

An `if` event input mapping essentially has an `if` expression that results in an event rather than a value. The conditions (`<expr>`) are guard expressions that evaluate to a boolean value. The guard expressions may be arbitrary expressions, similar to the use of expressions for values in output mappings.

If a condition holds, the corresponding event is chosen by the mapping. The entries are processed in the order they are specified: if the first guard (of the `if`) holds, the first event is chosen, otherwise if the second guard (of an `elif`) holds, the second event is chosen, etc. The `if` part is mandatory for an `if` event input mapping, but the `elif` parts are optional. If it has `elif` parts, it may have as many of them as necessary.

The last part of the `if` is allowed to be an `else`, to indicate that if none of the previous guards hold, the event of the `else` is to be chosen. Using an `else` entry is optional, but if used, there may only be one such entry, and it must be the last entry of the input mapping.

An `if` input mapping must be complete. If such a mapping has an `else`, it is always complete. If it doesn't have an `else`, at least one of its `if` or `elif` guards must hold. If none of the guards hold, the mapping results in a runtime error.

The event that is chosen by the mapping becomes enabled, and at most one transitions may be taken for it, like for single event input mappings.

It is also allowed to declare an optional local SVG file declaration, that only applies to that specific input declaration.

Update input mapping

With SVG input mappings it is also possible to update the values of input variables:

```
svgin id <id> do <updates>;
svgin id <id> do <updates> file <file>;
```

Compared to a single event input mapping or `if` event input mapping, no event is specified. Instead, updates can be specified, like on edges. Clicking on the interactive SVG element then leads to the updates being executed, rather than an event of the model being enabled.

It is also allowed to declare an optional local SVG file declaration, that only applies to that specific input declaration.

Be aware that the initial value of an input variable cannot be set in the specification. Instead, CIF tools like the CIF simulator allow to set the initial value via an option.

Uniqueness

Similar to output mappings, all input mappings must be unique, per SVG image. That is, no two input mappings for the same SVG image may use the same SVG element id.

Note that it is allowed to have an input mapping and an output mapping (or even multiple output mappings) for the same SVG element. It is also allowed for multiple SVG input mappings to enable the same event, or update the same input variables.

3.3. Text

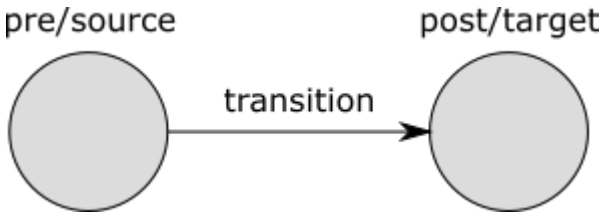
3.3.1. Print declarations

CIF supports custom textual output from CIF models by means of print output. See the CIF language tutorial for an introduction and examples. Here we discuss the details.

The core concept of print output is the *print declaration*. The print declarations allow specifying what to print. They also optionally allow specifying when to print it, how to print it, and to where to print it.

Transition centric view

Unlike many programming languages that have print statements, CIF has print declarations. Print declarations are declarative, rather than imperative. Print declarations center on transitions. The following is a schematic overview of a transition:



The transition is graphically depicted as an arrow, with a label. The label denotes the event that occurred, or `time` for time transitions. Every transition has a *pre* or *source* state, and a *post* or *target* state. The states are graphically depicted as circles, with a name above them. These names are only used so that they can be referred to from the text.

The figure thus contains the transition and its immediate environment (its source and target states). Print declarations allow specifying what, how, and when to print, for transitions. Since the transition is the central part, we call this the *transition centric view*.

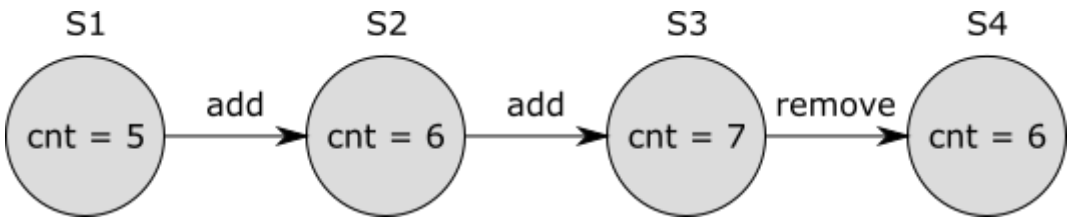
Extended trace

Conceptually, simulations and executions result in traces. A trace consists of the states that are encountered and the transitions that are taken, during the simulation or execution. For instance, consider the following CIF specification:

```
automaton buffer:
  event add, remove;
  disc int cnt = 5;

  location:
    initial;
    edge add    when cnt < 10 do cnt := cnt + 1;
    edge remove when cnt >  0 do cnt := cnt - 1;
end
```

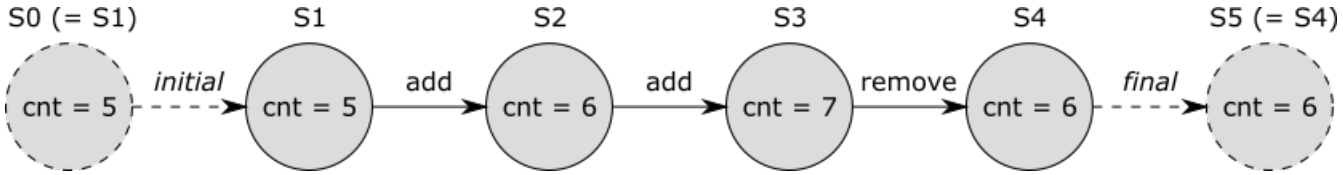
The `buffer` initially contains five items (`cnt = 5`). As long as there is room in the buffer (`cnt < 10`), an item can be added (event `add`) to the buffer. As long as the buffer is not empty (`cnt > 0`), an item can be removed (event `remove`) from the buffer. The following is a potential simulation trace:



In the initial state (S1), variable `cnt` has value `5`. An item is then added (transition for the event `add`), resulting in a new state (S2), where `cnt` has value `6`. After that, another item is added, leading to state S3, where `cnt` has value `7`. Finally, an item is removed by means of a transition for event `remove`, leading to state S4, where `cnt` has value `6` again.

As can be seen in the figure, the initial state (S1) has no incoming transition. Furthermore, the

last/final state (S4) has no outgoing transition. As we'll see later, that poses some difficulties in applying the transition centric view. To get around these limitations, the trace is extended as follows:



A transition with the *initial* label is added before the initial state, and a transition with the *final* label is added after the final state. This ensures that all states have an incoming and outgoing transition. These two added transitions are *virtual*, they don't actually exist. They are added only for the benefit of print output. That is, they don't affect the behavior of the model, and are thus not part of the 'real' trace.

The added 'initial' virtual transition has no source state, and the added 'final' virtual transition has no target state. Since in the transition centric view every transition should have a source and a target state, we add the initial state (S1) as source state (S0) for the 'initial' transition, and the final state (S4) as target state (S5) for the 'final' transition. This way, every transition (whether virtual or real) has a source and target state, and fits the transition centric view. For the 'initial' and 'final' virtual transitions, the source and target states are thus the same (S0 = S1, S4 = S5).

The trace with added virtual 'initial' and 'final' transitions, and with the duplicates (S0 and S5) of the source and final states, is called the *extended trace*.

Specifying the text to print

Every print declaration must specify *what* to print, using CIF expressions. These CIF expressions can for instance refer to variables from the CIF model. Since variables can change values, it is important to know in which state the expression is evaluated. In the transition centric view, we have two states to choose from, the pre/source state and the post/target state. The following variants are supported to specify what to print:

```
print EXPR;
print pre EXPR;
print post EXPR;
print pre EXPR post EXPR;
```

The first variant has only one expression, which is printed in the post/target state. The third variant is identical, but explicitly indicates that the post/target state is used. The second variant also has a single expression, but is evaluated in the pre/source state. The fourth variant has two expression, one of which is evaluated in the pre/source state, and one of which is evaluated in the post/target state. The first three variants only print one piece of text per transition, while the fourth variant prints two pieces of text per transition.

Consider the following examples:

```
print "a";
print pre x post x;
```

The first example prints `a` after each transition. The second example prints the value of variable `x` as it was before the transition, as well as the value of variable `x` as it is after the transition, for each transition.

Quoting and escaping

The expressions that indicate what text to print may be of any type, and the result of evaluating the expression is converted to a textual representation that closely resembles the textual syntax of CIF. For string literals this means that the text is escaped, and double quotes are added. If however the result of the expression is a `string` typed value, then that string is used 'as is' (without quoting, and without escaping). Thus, consider the following example:

```
print ["a\"b"];
print "a\"b";
```

The first print declaration prints a list that contains a single string value, while the second print declaration prints the single string value directly. This results in the following printed output:

```
["a\"b"]
a"b
```

That is, in general string values are quoted and escaped, as is the case for the first example, where the result is a list. However, if the entire result is a string, as is the case with the second example, the string value is used 'as is', without quoting and without escaping.

Transition filtering

It is possible to filter the transitions to which print declarations apply, by looking at the kind of the transitions (their labels). The following transition filters are available:

Filter	Matches	Does not match
`event`	All event transitions, including those for the `tau` event.	All time and virtual transitions.
`time`	All time transitions.	All event and virtual transitions.
e	All transitions for event `e`.	All time and virtual transitions, as well as all transitions for 'other' events (including those for the `tau` event).

Filter	Matches	Does not match
`initial`	The single virtual *initial* transition.	All event and time transitions, as well as the single virtual *final* transition.
`final`	The single virtual *final* transition.	All event and time transitions, as well as the single virtual *initial* transition.

Transition filters may be combined, leading to a combined transition filter that matches if any of the individual transition filters matches. The individual filters must be separated by commas.

Transition filtering is optional. If no transition filter is supplied, the default transition filter is used, which is `initial, event, time`. That is, by default output is printed for all transitions, except for the *final* virtual transition.

Consider the following examples:

```
print EXPR for e;
print EXPR for time, event;
print EXPR for initial, final;
print EXPR for machine1.start, machine2.start;
print EXPR;
```

The first example prints only for transitions for event `e`. That is, time transitions and virtual transitions, as well as transitions for all other events, are filtered out. The second example prints only for event and time transitions, and thus filters out the virtual transitions. The third example prints only for the virtual transitions, and thus filters out all 'real' transitions (the event and time transitions). The fourth example prints only for transitions for the `start` events declared in `machine1` and `machine2` (which could for instance be automata), and filters out time transitions, virtual transitions, and transitions for all other events. The fifth example does not specify a filter, and thus prints for the *initial* virtual transition, as well as all event and time transitions, but excludes the *final* virtual transition.

State filtering

It is possible to filter the transitions to which print declarations apply, by looking at the pre/source and post/target states of the transitions. The following variants are supported for state filtering:

```
print EXPR when PRED;
print EXPR when pre PRED;
print EXPR when post PRED;
print EXPR when pre PRED post PRED;
```

The first variant has only one predicate, which is evaluated in the post/target state. The third variant is identical, but explicitly indicates that the post/target state is used. For both these variants, text is only printed if the post/target state satisfies the given predicate. That is, if the predicate

evaluated in the post/target state of the transition results in value `true`, text may be printed. Otherwise, no text is printed for that transition.

The second variant also has a single predicate, but is evaluated in the pre/source state. Text is thus only printed if the pre/source state satisfies the given predicate.

The fourth variant has two predicates, one of which is evaluated in the pre/source state, and one of which is evaluated in the post/target state. Text is only printed if the pre/source state satisfies the first predicate, and the post/target state satisfies the second predicate. If either of the states does not satisfy its corresponding predicate, no text is printed for that transition.

Consider the following examples:

```
print "a";
print "b" when aut.loc;
print "c" when pre aut.loc;
print "d" when pre x mod 2 = 0 post x mod 2 = 1;
```

The first example prints `a` after each transition, and does not restrict the pre/source or post/target states. The second example only prints `b` for transitions that end in location `loc` of automaton `aut`. The third example only prints `c` for transitions that start in location `loc` of automaton `aut`. The fourth example only prints `d` for transitions where the value of variable `x` is even before the transition, and odd after the transition.

State filtering is thus optional. If no pre/source state predicate is specified, the default is `true`. Similarly, if no post/target state predicate is specified, the default is `true` as well.

Combining filters

The transition and state filters are all optional. If however both a transition filter and a state filter are specified, they must both match in order for text to be printed. Consider the following examples:

```
print time for time when time > 3;
print x for e when pre x < 3 post x > 5;
print pre y post y when pre y = 5;
```

For the first example, the value of variable `time` is printed after time transitions, if after those time transitions the value of variable `time` is strictly larger than three. For the second example, the value of variable `x` is printed after transitions for event `e` if before such transitions the value of variable `x` is strictly less than three, and after such transitions the value of variable `x` is strictly larger than `5`. For the third example, the value of variable `y` is printed twice for all 'default' transitions, once in the pre/source state, and once in the post/target state, but only if the value of variable `y` before the transition is exactly five. So if the condition over the pre/source state does not hold, the value of variable `y` is not printed, not for the pre/source state, and also not for the post/target state.

Output file/target

By default text is printed to the console, to the standard output stream, usually called 'stdout'. It is however possible to explicitly specify the file or special target to which the output is to be printed. Consider the following examples:

```
print ... file "txt_files/some_file.txt";
print ... file "../files/some_file.txt";
```

For the first print declaration, the text is printed to a file named `some_file.txt` in the `txt_files` directory, where the `txt_files` directory is located in the same directory as the CIF file. For the second print declaration, the text is printed to a file named `some_file.txt` in the `files` directory, where the `files` directory is located in the parent directory of the directory that contains the CIF file.

Files and directories are separated using slashes (`/`), regardless of the used operating system. It is also allowed to use backslashes (`\`), which are more common on Microsoft Windows operating systems, but they need to be escaped as `\\` for this to work, making it easier to use slashes (`/`) instead.

The following special non-file targets are supported:

- `":stdout"`: prints to the standard output stream. In the Eclipse ESCET IDE, the text ends up on the console, as black text.
- `":stderr"`: prints to the standard error stream. In the Eclipse ESCET IDE, the text ends up on the console, as red text.

Besides these local file indications, CIF also supports global print file declarations, which influence the default, in case no local file or special target is specified.

Scoping

Print declarations can be specified in all components. That is, they can be specified in the top level scope of the specification, in groups and group definitions, as well as in automata and automaton definitions.

3.3.2. Print file declarations

Print declarations can specify the file or special target to which text is to printed. If they don't specify anything, the default is to print to the standard output stream, usually called 'stdout'. This default can be influenced by means of *print file declarations*, which act as a sort of global declaration of the file or special target. A typical print file declaration looks like this:

```
printfile "some_file.txt";
```

Paths

It is possible to use relative (or absolute) paths for the files. For instance:

```
printfile "txt_files/some_file.txt";
printfile "../files/some_file.txt";
```

For the first print file declaration, text is printed to a file named `some_file.txt` in the `txt_files` directory, where the `txt_files` directory is located in the same directory as the CIF file. For the second print file declaration, text is printed to a file named `some_file.txt` in the `files` directory, where the `files` directory is located in the parent directory of the directory that contains the CIF file.

Files and directories are separated using slashes (`/`), regardless of the used operating system. It is also allowed to use backslashes (`\`), which are more common on Microsoft Windows operating systems, but they need to be escaped as `\\` for this to work, making it easier to use slashes (`/`) instead.

Special targets

The following special non-file targets are supported:

- `":stdout"`: prints to the standard output stream. In the Eclipse ESCET IDE, the text ends up on the console, as black text.
- `":stderr"`: prints to the standard error stream. In the Eclipse ESCET IDE, the text ends up on the console, as red text.

Scoping

Print file declarations can (just as print declarations) be specified in all components. That is, they can be specified in the top level scope of the specification, in groups and group definitions, as well as in automata and automaton definitions. A print file declaration that is specified in a certain scope applies to that scope, as well as all child scopes that don't have a print file declaration of their own. Additionally, print file declarations apply to all print declarations without local file declarations, in scopes to which the print file declarations apply.

For instance, consider the following CIF specification:

```
printfile "file1.txt";
```

```
print 5;
print 6 file "file2.txt";

group g:
  printfile "file3.txt";

  print 7;
  print 8;
  print 9 file "file4.txt";

  group h:
    printfile "file5.txt";
  end

  group i:
    print 10;
  end
end
```

The print declaration that prints `5` is declared in the top level scope of the specification, and does not have a local file declaration. As such, the print file declaration from the top level scope (`file1.txt`) applies to it. The print declaration that prints `6` is declared in the same scope, but has a local print file declaration (`file2.txt`). The local print file declaration (`file2.txt`) overrides the print file declaration from the top level scope (`file1.txt`).

Group `g` also has a print file declaration (`file3.txt`). This overrides the print file declaration from the top level scope (`file1.txt`). The print file declaration from group `g` (`file3.txt`) thus applies to the print declarations that print `7` and `8`, as they are declared in the same scope, and don't have local print file declarations. The print declaration that prints `9` has a local print file declaration (`file4.txt`), and as such the print file declaration from group `g` (`file3.txt`) does not apply to it.

The print file declaration (`file5.txt`) from group `g.h` does not apply to any print declarations. No text is thus printed for it.

Group `g.i` does not have a print file declaration, so the print file declaration (`file3.txt`) from group `g` (the parent scope of group `g.i`) applies to the print declaration that prints `10`.

See also the Push print file declarations into print declarations CIF to CIF transformation.

Uniqueness

In every scope (or component), in principle at most one print file declaration may be specified. The exception is that for a single scope, specified in multiple CIF files (when imports are used), if the print file declarations refer to the same target, they are merged. If however the two CIF files refer to different targets, this is still considered an error. It is also considered an error if two print file declarations are present in a single scope, in a single file, regardless of whether they refer to the same target or not. In the end, after processing imports, for every scope there must be a unique

print file declaration, if one is declared at all.

If the same file or special target is used/specified multiple times, in different print file declarations or locally in print declarations, the text produced by all these print declarations is all printed to that single file or special target. For instance, consider the following CIF specification:

```
print 5 file "some_file.txt";
print 6 file "some_file.txt";
```

Even though both print declarations specify a file locally, they use the same file. The output of both print declarations is thus printed to the same file.

3.3.3. Text formatting details

For text formatting, CIF features the `fmt` standard library function. This page describes all the details of using that function. For more introductory information and examples of the use of the `fmt` function, in the context of print output, see the language tutorial's lesson on text formatting.

Format patterns

The `fmt` function requires at least one argument, the *format pattern*. The remaining arguments of the function are the *values* that can be inserted into the format pattern, to allow variable output. For instance, consider:

```
fmt("%s %s", x, y)
```

Here, `"%s %s"` is the format pattern, `x` is the first value, and `y` is the second value.

The first argument, the format pattern, must be a string literal. That is, it must be text between double quotes (").

The real usefulness of format patterns comes from the inclusion of the values into the format pattern. The values can be included by inserting format specifiers (e.g. `%s`) into the format pattern. Multiple values may be used by including multiple format specifiers. The first format specifier includes the first value, the second format specifier includes the second value, etc.

The result of `fmt` applied to it arguments is the text of the format pattern, with the format specifiers replaced by their corresponding values.

Format specifiers can be customized to influence how their corresponding values are to be inserted. For instance, consider:

```
fmt("%s %.2e", x, y)
```

Then, assuming variable `x` has value `3` and variable `y` has value `5.6`, the result of the `fmt` function is `3`

`5.60e+00`.

Specifiers

Format patterns support various types of format specifiers, each performing a different kind of conversion on its corresponding value. The following table lists the available conversions, the types of values that they support, and a short description of the output of the conversion:

Conversion	Supported types	Output
%b	`bool`	Either `true` or `false`.
%B	`bool`	Same as `%b`, but in upper case.
%d	`int`	Decimal integer notation.
%x	`int`	Hexadecimal integer notation, using `0-9` and `a-f`.
%X	`int`	Hexadecimal integer notation, using `0-9` and `A-F`.
%e	`real`	Computerized scientific notation, using `e`.
%E	`real`	Computerized scientific notation, using `E`.
%f	`real`	Decimal number notation.
%g	`real`	General scientific notation, using `e`.
%G	`real`	General scientific notation, using `E`.
%s	any type	General textual representation.
%S	any type	General textual representation, in upper case.

For the `%d`, `%x`, and `%X` specifiers, `int` typed values are supported, as are integer types with ranges (e.g. `int[0..5]`). The `%s` and `%S` specifiers support values of any type.

The output of the `%B` specifier is identical to the output of the `%b` specifier, where all letters in the output are converted to upper case letters. This duality (lower case specifier conversion versus upper case specifier conversion) is present in all conversions that can have letters in their output.

Specifier syntax

Specifiers can be customized to influence the conversion of their corresponding values to text. The general syntax of specifiers is:

```
%[value_index$][flags][width][.precision]conversion
```

All specifiers start with a percentage character (`%`).

They are optionally followed by a value index. If the value index is specified, it must be followed by a dollar sign (`$`). The value index is a positive decimal integer indicating the position of the value in

the value list. The first value is referenced by `1$`, the second by `2$`, etc. Index zero and indices starting with zero are not allowed.

The index is followed by optional flags. Flags are characters that modify the output, and may be specified in any order. Each flag may only be specified once per specifier. The set of valid flags depends on the conversion.

After the flags, an optional width may be specified. The width is a non-negative decimal integer indicating the minimum number of characters that should be included in the result, for that specific specifier.

After the width, a precision may optionally be specified. A precision is always preceded by a dot (`.`). The precision is a non-negative decimal integer used to restrict the number of characters. The specific behavior depends on the conversion.

Specifiers always end with the character indicating the conversion to perform.

Implicit and explicit indexing

By default, format specifiers are processed in order. The first specifier then implicitly uses the first value, the second specifier implicitly uses the second value, etc. However, if an explicit value index is given, that explicit index indicates the value to use, and the specifier with the explicit index does not influence subsequent implicit indexing. That is, consider the following:

```
fmt("%s %1$s %3$f %d %f %1$s", 1, 2, 3.0);
```

The first specifier (`%s`) does not specify an explicit index, and thus implicitly uses the first value. The second specifier (`%1$s`) explicitly specifies index `1` and thus uses the first value. The third specifier (`%3$f`) explicitly specifies index `3` and thus uses the third value. The fourth specifier (`%d`) is the second specifier to not explicitly specify an index, and thus implicitly uses the second value. The fifth specifier (`%f`) is the third specifier to not explicitly specify an index, and thus implicitly uses the third value. Finally, the sixth specifier (`%1$s`) explicitly specifies index `1`, and thus uses the first value. The result of the formatting is:

```
1 1 3.000000 2 3.000000 1
```

Flags

The following flags are available:

Flag	b/B	d	x/X	e/E	f	g/G	s/S	Effect
-	yes	yes	yes	yes	yes	yes	yes	The result will be left-justified.
+	no	yes	no	yes	yes	yes	no	The result will always include a sign.

Flag	b/B	d	x/X	e/E	f	g/G	s/S	Effect
(space)	no	yes	no	yes	yes	yes	no	The result will include a leading space for non-negative values.
`0`	no	yes	yes	yes	yes	yes	no	The result will be zero-padded.
`,`	no	yes	no	no	yes	yes	no	The result will include commas (`,`) as grouping separator.

The first column shows the available flags, the minus (`-`), the plus (`+`), the space, the zero (`0`), and the comma (`,`). The middle columns indicate for each of the different conversions, whether the flags are supported. The last column gives short descriptions of the effects of the flags.

The `-` flag can be used to left-justify the text in case a width is used, as the default is to right-justify. It is supported by all format specifiers. The `-` flag requires the use of a width, and can not be combined with the `0` flag.

The `+` flag can be used to always include a sign. It only applies to certain numeric format specifiers. That is, by default non-negative numbers don't have a sign, and negative numbers start with a `-` character. By including the `+` flag, non-negative numbers start with a `+` character, and negative numbers still start with a `-` character. The `+` and space flags can not be combined.

The space flag can be used force a leading space to be included for non-negative values. It only applies to certain numeric format specifiers. That is, by default non-negative numbers don't have a sign, and negative numbers start with a `-` character. By including the space flag, non-negative numbers start with a space character, and negative numbers still start with a `-` character. The `+` and space flags can not be combined.

The `0` flag can be used to zero pad numbers, in case a width is used, the text is right-justified, and the text is shorter than the width. It only applies to certain numeric format specifiers. The `0` flag requires the use of a width, and can not be combined with left-justification (the `-` flag).

The `,` flag can be used to include commas (`,`) as grouping separators. It only applies to certain numeric format specifiers. That is, by default numbers are just a sequence of digits. By using the `,` flag, longer numbers are placed in groups of thousands and the 'thousands separator' (the `,` character) occurs every three digits. For instance, `12345678` would then become `12,345,678`.

Further details on the effects of flags are given in the sections describing the individual conversions.

Width

The width is a non-negative decimal integer indicating the minimum number of characters that should be included in the result, for that specific specifier. If no width is given, there is no minimum number of characters.

If the textual representation of the value is shorter than the width, the text is right-justified. That is, the text will be padded by spaces to the left of the text until the total number of characters equals the width. To left-justify the text, use the `-` flag in combination with a width. This results in padding

with spaces to the right of the text.

If the `0` flag is specified, and the textual representation of the value is shorter than the width, `0` padding is used (to the left, as the `-` flag may not be used when the `0` flag is used). See the sections describing the individual number related specifiers for further details.

If the text is longer than the width, the whole text is included in the result (it is not truncated), and the width essentially has no effect.

`%b` and `%B` specifiers

The `%b` and `%B` specifiers convert boolean values to text. The specifiers only support boolean values. The specifiers support explicit indices, widths, and the `-` flag. They don't supports any of the other flags, and precisions.

The resulting text is constructed as follows:

- Value `true` results in the text `true`, and value `false` results in the text `false`.
- For the `%B` specifier, the text is converted to upper case, i.e. `TRUE` and `FALSE` respectively.
- If a width is specified, and the resulting text so far is shorter than that width, spaces are added until the given width is reached. If the `-` flag is given, spaces are added to the right, otherwise spaces are added to the left.

Here are some examples, assuming `b` has value `true`:

```
fmt("%b", b);       // true
fmt("%B", b);       // TRUE
fmt("_%10b_", b);   // _      true_
fmt("_%-10b_", b);  // _true      _
```

`%d` specifier

The `%d` specifier converts integer numbers to text, using decimal representations of integers. The specifier only supports integer values (`int` typed values, and values of integer types with ranges, such as `int[0..5]`). The specifier supports explicit indices, widths, and all flags. It doesn't support precisions.

The resulting text is constructed as follows:

- The resulting text is initialized to the decimal representation of the absolute value of the integer number. It thus consists of only the digits `0` through `9`.
- If the `,` flag is given, thousand separators (`,` characters) are inserted as needed, for longer numbers.
- If the number is negative, it is prefixed with a `-` character.

- If the number is non-negative, and the space flag is given, a space is added before the text.
- If the number is non-negative, and the `+` flag is given, a `+` character is added before the text.
- If a width is given, the text so far is shorter than the width, and the `0` flag is given, then `0` characters are added before the other digits, and after any non-digit characters, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and the `-` flag is given, then space characters are added before the result, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and neither the `0` flag nor the `-` flag is given, then space characters are added after the result, until the desired width is reached.

Here are some examples, assuming `x` has value `12345` and `y` has value `-2345`:

```
fmt("%d", x);           // 12345
fmt("%,d", x);          // 12,345
fmt("_%10d_", x);       // _     12345_
fmt("_%-10d_", x);      // _12345     _
fmt("_%0,10d_", x);     // _000012,345_
fmt("_%- 10d_", x);     // _ 12345    _
fmt("_%- 10d_", y);     // _-2345     _
fmt("_%-+10d_", x);     // _+12345    _
fmt("_%-+10d_", y);     // _-2345     _
fmt("_%3d_", x);        // _12345_
```

`%x` and `%X` specifiers

The `%x` and `%X` specifiers convert integer numbers to text, using hexadecimal representations of integers. The specifiers only support integer values (`int` typed values, and values of integer types with ranges, such as `int[0..5]`). The specifiers support explicit indices, widths, and the `-` and `0` flags. They don't supports any of the other flags, and precisions.

The resulting text is constructed as follows:

- The signed integer number in range [-2,147,483,648 .. 2,147,483,647] is first converted to an unsigned integer number in range [0 .. 4,294,967,295]. That is, for negative numbers, 2^{32} is added.
- The resulting text is initialized to the hexadecimal representation of the unsigned integer number. It thus consists of the digits `0` through `9` and letters `a` through `f`.
- For the `%X` specifier, the text is converted to upper case, i.e. letters `a` through `f` are converted to letters `A` through `F`.
- If a width is given, the text so far is shorter than the width, and the `0` flag is given, then `0` characters are added before the result, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and the `-` flag is given, then space characters are added before the result, until the desired width is reached.

- If a width is given, the text so far is shorter than the width, and neither the `0` flag nor the `-` flag is given, then space characters are added after the result, until the desired width is reached.

Here are some examples, assuming `a` has value `5543` and `b` has value `-1`:

```
fmt("%x", a);           // 15a7
fmt("0x%x", a);         // 0x15a7
fmt("0x%X", a);         // 0x15A7
fmt("0x%X", b);         // 0xFFFFFFFF
fmt("_%10x_", a);       // _      15a7_
fmt("_%-10x_", a);      // _15a7      _
fmt("_%3x_", a);        // _15a7_
```

`%e` and `%E` specifiers

The `%e` and `%E` specifiers convert real numbers to text, using computerized scientific notation. The specifiers only support real values. The specifiers support explicit indices, widths, the `-`, `+`, space, and `0` flags, and precisions. They don't supports any of the other flags.

Real numbers include a *decimal mark*, a symbol used to separate the integer part from the fractional part of number, when written in decimal form. This decimal mark is denoted by a dot (`.`).

The resulting text is constructed as follows:

- The decimal mark of the real number is shifted to ensure that at most one non-zero digit occurs to the left of it. That is, for real numbers `0.012`, `0.12`, `1.2`, `12.0`, and `120.0`, the decimal mark is shifted -2, -1, 0, 1, and 2 digits to the left, respectively. This results in the following real numbers: `1.2`, `1.2`, `1.2`, `1.2`, and `1.2`. For zero, the decimal mark is not shifted.
- The single decimal digit before the decimal mark is included in the result. If there is no digit before the decimal mark (in case the real number is zero), a single `0` digit is included in the result.
- If the precision is specified and is not zero, or if the default precision is used, a dot (`.`) is added after the single digit.
- The digits after the decimal mark are added after the dot. Exactly 'precision' digits will be added. If no precision is specified, it defaults to `6` digits after the dot. If not enough digits are available after the dot, additional `0` characters are added after them, to reach the desired precision. If too many digits are available after the dot, digits are removed from the right until the desired precision is reached. Rounding using the 'half up' algorithm is used to ensure correct results in case digits are removed.
- If the `%e` specifier is used, an `e` character is added to the end. If the `%E` specifier is used, an `E` character is added to the end.
- A sign character is added after that. That is, if the decimal mark was shifted a negative number of digits to the left, a `-` character is added, otherwise a `+` character is added.
- The number of digits that was shifted is added as a decimal number, at the end. At least two

decimal digits are added, so if the number of digits that was shifted can be represented using a single decimal digit, a `0` is added between the `e` or `E` character and the number of digits that was shifted.

- If the real number is negative, the text is prefixed with a `-` character.
- If the real number is non-negative, and the space flag is given, a space is added before the text.
- If the real number is non-negative, and the `+` flag is given, a `+` character is added before the text.
- If a width is given, the text so far is shorter than the width, and the `0` flag is given, then `0` characters are added before the other digits, and after any non-digit characters, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and the `-` flag is given, then space characters are added before the result, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and neither the `0` flag nor the `-` flag is given, then space characters are added after the result, until the desired width is reached.

Here are some examples, assuming `a` has value `12345.6789` and `b` has value `-0.00002345678`:

```
fmt("%e", a);           // 1.234568e+04
fmt("%E", a);           // 1.234568E+04
fmt("%.3e", a);         // 1.235e+04
fmt("%.3e", b);         // -2.346e-05
fmt("_%20e_", a);       // _         1.234568e+04_
fmt("_%-20e_", a);      // _1.234568e+04        _
fmt("_%5e_", a);        // _1.234568e+04_
fmt("_%020e_", a);      // _000000001.234568e+04_
fmt("_%-+20e_", a);     // _+1.234568e+04       _
fmt("_%-+20e_", b);     // _-2.345678e-05       _
fmt("_%- 20e_", a);     // _ 1.234568e+04       _
fmt("_%- 20e_", b);     // _-2.345678e-05       _
```

`%f` specifier

The `%f` specifier converts real numbers to text, using a decimal number notation. The specifier only supports real values. The specifier supports explicit indices, widths, all flags, and precisions. That is, all features of format specifiers are supported.

Real numbers include a *decimal mark*, a symbol used to separate the integer part from the fractional part of number, when written in decimal form. This decimal mark is denoted by a dot (`.`).

The resulting text is constructed as follows:

- The decimal digits before the decimal mark are included in the result. If there are no digits before the decimal mark, a single `0` digit is included in the result.
- If the `,` flag is given, thousand separators (`,` characters) are inserted as needed, for longer numbers.

- If the precision is specified and is not zero, or if the default precision is used, a dot (`.`) is added after the digits.
- The digits after the decimal mark are added after the dot. Exactly 'precision' digits will be added. If no precision is specified, it defaults to `6` digits after the dot. If not enough digits are available after the dot, additional `0` characters are added after them, to reach the desired precision. If too many digits are available after the dot, digits are removed from the right until the desired precision is reached. Rounding using the 'half up' algorithm is used to ensure correct results in case digits are removed.
- If the real number is negative, the text is prefixed with a `-` character.
- If the real number is non-negative, and the space flag is given, a space is added before the text.
- If the real number is non-negative, and the `+` flag is given, a `+` character is added before the text.
- If a width is given, the text so far is shorter than the width, and the `0` flag is given, then `0` characters are added before the other digits, and after any non-digit characters, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and the `-` flag is given, then space characters are added before the result, until the desired width is reached.
- If a width is given, the text so far is shorter than the width, and neither the `0` flag nor the `-` flag is given, then space characters are added after the result, until the desired width is reached.

Here are some examples, assuming `a` has value `12345.6789` and `b` has value `-0.00002345678`:

```
fmt("%f", a);           // 12345.678900
fmt("%f", b);           // -0.000023
fmt("%.3f", a);         // 12345.679
fmt("_%20f_", a);       // _         12345.678900_
fmt("_%-20f_", a);      // _12345.678900        _
fmt("_%-,20f_", a);     // _12,345.678900       _
fmt("_%5f_", a);        // _12345.678900_
fmt("_%020f_", a);      // _0000000012345.678900_
fmt("_%-+20f_", a);     // _+12345.678900       _
fmt("_%-+20f_", b);     // _-0.000023           _
fmt("_%- 20f_", a);     // _ 12345.678900       _
fmt("_%- 20f_", b);     // _-0.000023           _
```

`%g` and `%G` specifiers

The `%g` and `%G` specifiers convert real numbers to text, using general scientific notation. The specifiers only support real values. The specifiers support explicit indices, widths, all flags, and precisions. That is, all features of format specifiers are supported.

Real numbers include a *decimal mark*, a symbol used to separate the integer part from the fractional part of number, when written in decimal form. This decimal mark is denoted by a dot (`.`).

If the real number is greater than or equal to 10^{-4} but less than $10^{\text{precision}}$, the result is as `%f`.

Otherwise, the result is as `%e` and `%E`, for `%g` and `%G` respectively. However, the total number of significant digits of the result is equal to the precision of the `%g` or `%G` specifier, and defaults to `6` if not specified. If the specified precision is `0`, it is taken to be `1` instead.

Here are some examples, assuming `a` has value `12345.6789` and `b` has value `-0.00002345678`:

```
fmt("%g", a);           // 12345.7
fmt("%g", b);           // -2.34568e-05
fmt("%G", b);           // -2.34568E-05
fmt("%.3g", a);         // 1.23e+04
fmt("_%20g_", a);       // _              12345.7_
fmt("_%-20g_", a);      // _12345.7              _
fmt("_%-,20g_", a);     // _12,345.7             _
fmt("_%5g_", a);        // _12345.7_
fmt("_%020g_", a);      // _00000000000012345.7_
fmt("_%-+20g_", a);     // _+12345.7             _
fmt("_%-+20g_", b);     // _-2.34568e-05         _
fmt("_%- 20g_", a);     // _ 12345.7             _
fmt("_%- 20g_", b);     // _-2.34568e-05         _
```

`%s` and `%S` specifiers

The `%s` and `%S` specifiers convert any value to text. The specifiers support values of any type. The specifiers support explicit indices, widths, and the `-` flag. They don't supports any of the other flags, and precisions.

The resulting text is constructed as follows:

- If the value has a `string` type, the value of that string is used as is, without escaping and double quoting. Otherwise, the value is converted to a textual representation closely resembling the CIF textual syntax (ASCII syntax), with string values surrounded by double quotes, and with special characters (tabs, new lines, double quotes, and backslashes) escaped.
- For the `%S` specifier, the text is converted to upper case, i.e. letters `a` through `z` are converted to letters `A` through `Z`.
- If a width is specified, and the resulting text so far is shorter than that width, spaces are added until the given width is reached. If the `-` flag is given, spaces are added to the right, otherwise spaces are added to the left.

Here are some examples, assuming `x` has value `"aBcD"`:

```
fmt("%s", x);       // aBcD
fmt("%S", x);       // ABCD
fmt("_%10s_", x);   // _      aBcD_
fmt("_%-10s_", x);  // _aBcD      _
```

As indicated above, `string` typed values are of special interest. For instance, consider the following examples:

```
fmt("a %s b", ("some \\ text", 6));     // a ("some \\ text", 6) b
fmt("a %s b", "some \\ text");          // c some \ text d
```

The first example has a tuple as value, with a string and an integer number. The output of this example is `a ("some \\ text", 6) b`. Note how the string in the tuple is included in the output with double quotes around it, and how the escaped backslash is also escaped in the resulting text.

The second example has a value that is a string directly (e.g. not contained in a tuple or some other container). This mapping results in `c some \ text d`. Note how the string is included in the output 'as is', i.e. it is not surrounded by double quotes, and the backslash is not escaped.

In general, when using a `%s` specifier, string values are double quoted and escaped. If however the entire value is a string, then that string is used 'as is' (without quoting, and without escaping).

Also of special interest are real values. The number of digits after the decimal point can vary. For instance, consider the following examples:

```
fmt("a %s b", 1 / 3);       // a 0.3333333333333333 b
fmt("c %s d", 1 / 2);       // c 0.5 d
```

For more control over the number of digits after the decimal point, use one of the floating point number format specifiers: %e, %E, %f, %g, or %G.

Escaping and quoting

A double quote terminates a string literal, and thus also a format pattern, as format patterns are string literals. It is therefore not possible to include a double quote in the text of a format pattern without escaping it. That is, use `\"` inside a format pattern to include a single double quote. There are other escape sequences as well. Use `\\` to include a single backslash (\), `\n` to include a single new line (to continue on a new line), and `\t` to include a tab character (for alignment).

Format specifiers start with a percentage character (`%`). If a percentage character is to be included as a percentage character instead of being interpreted as a format specifier, it needs to be escaped as well. That is, to include a percentage character as a percentage character, use `%%` in the format pattern.

For instance, the format pattern `"a\"b\nc\td\\e%%f"` results in the following text:

```
a"b
c       d\e%f
```

Unused values

The format pattern is automatically checked for unused values. That is, if a value is not used in the format pattern, there is no use in specifying it, and a warning will indicate this.

3.4. Annotations

3.4.1. Annotations

Annotations can be added to elements of a CIF specification, to annotate them with extra structured information. Tools that take CIF specifications as input can process the annotations that are attached to elements of the specification, and use the supplied information. Different tools may support different annotations. For the tools in the CIF toolset, the tool documentation indicates for each tool which annotations it supports and how it uses them.

For basic information on annotations, see the language tutorial. For a complete list of the annotations that are bundled with CIF, see the built-in annotations overview.

Syntax

Annotations are optional. If specified, annotations are indicated by a regular annotation name (for most elements of specifications) or double at-sign annotation name (for some elements of specifications). See below for further details on which elements require which kind of annotation name.

Optionally, the annotation may have arguments. If the annotation has arguments, these are put between parentheses, `(` and `)`. Parentheses may be used if there are no arguments, but this is not recommended. Multiple arguments are separated by commas. An extra comma may optionally follow the last argument, which can make it easier to comment out arbitrary arguments, while still maintaining a valid model.

Each argument consists of an optional name and a mandatory value. If both a name and a value are provided, they are separated by a `:` sign. If the argument has a name, it is either a CIF identifier or a CIF relative name. If the argument name is an identifier that is a CIF keyword, it needs to be escaped by prefixing it with a `$` sign. Similarly, identifiers within relative names need to be escaped as well. For instance, argument name `automaton` needs to be escaped as `$automaton`, and argument name `some.automaton` needs to be escaped as `some.$automaton`. The value of the argument is a CIF expression.

As an example of the syntax, consider the following input variable annotated with 7 annotations:

```
@anno
@some:anno
@anno1()
@anno2(arg: 3, false)
@anno3($input: true)
@anno4(some_arg: 3, some.other.arg: [123, 467])
```

```
@anno5(
    arg1: -sqrt(2),
    arg2: 3 * 9,
    arg3: "some text",
)
input bool x;
```

Some observations:

- The first three annotations do not have any arguments, the other four do.
- The arguments have values of various types.
- For `anno2`, the first argument is named `arg`, while the second argument is unnamed.
- For `anno3`, the `input` argument name is escaped to `$input` as `input` is a keyword.
- For `anno4`, the second argument uses a relative name as argument name.
- For `anno5`, the first and second argument have a computation as value.
- For `anno5`, there is a comma after each argument, including after the last argument. This makes it easy to later comment out any of the three arguments, or add an extra argument without having add an extra comma.

Placement

Annotations are placed before the elements that they annotate. Here is an example, where an input variable is annotated with an annotation without arguments:

```
@anno
input bool x;
```

For a specification, the annotations must be before any other elements of the specification. Here is an example, where the specification with the annotated input variable is extended with two annotations on the specification itself:

```
@@anno1(arg1: 5)
@@anno2(arg2: "some text")

@anno
input bool x;
```

Annotations are only supported for the following elements in CIF specifications. Most require regular annotation names, while annotations on specifications and annotations on elements of locations require double at-sign annotation names:

Element	**Annotation name**
Algebraic parameters	Regular

Element	Annotation name
Algebraic variables	Regular
Automata	Regular
Automaton definitions	Regular
Automaton instantiations	Regular
Constants	Regular
Continuous variables	Regular
Discrete variables	Regular
Edges	Double at-sign
Enumeration literals	Regular
Enumerations	Regular
Events	Regular
Function parameters	Regular
Function variables	Regular
Functions	Regular
Group definitions	Regular
Group instantiations	Regular
Groups	Regular
Input variables	Regular
Invariants	Regular (for invariants in components) Double at-sign (for invariants in locations)
Locations of automata	Regular
Specifications	Double at-sign
Type declarations	Regular

Constraints

The following general constraints apply to all annotations:

- Annotations are optional.
- An element of the specification may have multiple annotations.
- If an element has multiple annotations, different annotations can be put in any order.
- In case a single element is annotated with the same annotation multiple times, then the order of these annotations may be relevant, although this could differ per annotation.
- Each named argument of an annotation must have a different name. An annotation may also have any number of unnamed arguments.
- The order of the arguments of an annotation may be relevant. This can be different for each annotation.

Each annotation must be registered. When registering an annotation, an annotation provider is registered with it:

- If an annotation is used in a specification, but is not registered, a warning will indicate the unregistered annotation. If a tool supports an annotation, it or one of its dependencies should register the annotation. Hence, if an annotation is not registered, it will likely be ignored by all available tools.
- Annotations are an extension mechanism, and anyone can define and register their own annotations. Different CIF installations may therefore have different registered annotations. A CIF specification may thus have a warning in one installation, if the annotation is for instance not registered in that installation, while in another installation the annotation does not have a warning, as there it is registered.
- If an annotation is registered, it must be properly registered and its annotation provider must be properly implemented. See below for details on annotation providers.

Each annotation provider may impose additional constraints on the annotation it handles:

- An annotation and its arguments must satisfy any additional constraints imposed by the corresponding annotation provider.

  An annotation provider could for instance limit the elements that may be annotated with the annotation, require all elements of a certain type to be annotated if at least one of them is annotated, or only allow a single such annotation on each annotated element. It could also constrain certain arguments, or combinations of arguments, to be mandatory or optional, or require them to be named or unnamed. Furthermore, it could require arguments to have values of a certain type, to be statically evaluable, and so on. Different people could define the same annotation in different ways, such that their own installations interpret that annotation differently and impose different constraints for it. Hence, a specification with such annotations may be valid in one installation, and invalid in another.

- Annotations are an extension mechanism, and anyone can define and register their own annotations. Different CIF installations may therefore have different annotation providers for the same annotation, which may interpret the annotation differently and may impose different constraints for it. Hence, a specification with such annotations may be valid in one installation, and invalid in another.

Naming annotations

Annotations are an extension mechanism, and anyone can define and register their own annotations. Without rules, different people may define the same annotation in different conflicting ways. Therefore, we list some guidelines for naming of annotations. These guidelines are not strict rules and likely will not prevent all issues, but adhering to them will likely at least reduce such issues.

Consider the following guidelines:

- Name an annotation based on the conceptual information it represents, not on how it is used. The annotation can then be reused for other uses. This avoids having multiple annotations with the same information for different uses.

- Avoid naming an annotation after a specific tool that uses it. The annotation can then be reused by other tools. This avoids having multiple annotations with the same information or purpose for different tools.
- Annotations are given a name that is as short as possible, to make it easier to write them. For instance, `doc` for a documentation annotation, rather than `documentation`.
- By default, annotation names consist of only a single identifier. If the annotation is specific to a certain purpose, and chances are that similarly named annotations will be introduced in the future, leading to naming conflicts, the annotation name can be scoped. For instance, for annotations related to linking CIF specifications to Programmable Logic Controllers (PLCs), the annotation names may be prefixed with `plc`. For instance, `plc:input` could be the name of a PLC input annotation. This allows other `input` annotations for different purposes to be added as well.
- More than two levels can be used in annotation names if needed to distinguish the annotations and ensure they are unique.
- Annotation names are ideally lower case, consisting of multiple parts separated by colons, rather than having parts that consist of multiple words. For instance, use `plc:input` rather than `plcInput` or `plc_input`.
- If you add a custom annotation, that is not a built-in annotation delivered along with CIF itself, start the annotation with your company name. This prevents conflicts with built-in annotations, and with annotations provided by other companies. For instance, use `<company_name>:some:anno`.

Annotation providers

Annotations can be registered using the `org.eclipse.escet.cif.annotations` extension point, provided by the `org.eclipse.escet.cif.typechecker` plugin. In your own plugin, to register an annotation:

- Open the plugin's manifest (`META-INF/MANIFEST.MF` file).
- Navigate to the *Extensions* tab.
- Click the *Add* button.
- Uncheck the *Show only extension points from the required plug-ins* checkbox.
- Select *org.eclipse.escet.cif.annotations* from the list and click *Finish*.
- If and when asked whether to add the `org.eclipse.escet.cif.typechecker` plugin as a dependency, click *Yes*.
- On the *Extensions* tab, a dummy *org.eclipse.escet.cif.annotations* entry is automatically added, and it is selected.
- Click *Show extension point description* for more information on how to properly use the extension point.
- You can edit the extension on the *Extensions* tab using the UI, or navigate to the `plugin.xml` tab of the manifest editor and edit it there textually.

A proper extension to register an annotation will look something like this:

```
<extension
```

```
      point="org.company.cif.annotations">
   <provider
         annotationName="some:thing"
         class="org.company.cif.annotations.SomeThingAnnotationProvider"
         plugin="org.company.cif.annotations">
   </provider>
 </extension>
```

Some additional constraints must be adhered to:

- At most one annotation provider must be registered for each annotation name.
- If an annotation is registered, it must specify an annotation provider. It must indicate the plugin where the provider is located as well as the Java class that implements it.
- If an annotation is registered, the registered annotation provider class must exist within that plugin, and the CIF type checker must be able to find and load it.
- If an annotation is registered, the registered annotation provider class must extend the `org.eclipse.escet.cif.typechecker.annotations.AnnotationProvider` class. The `AnnotationProvider` class has additional information and requirements. Consult its JavaDoc for the details.

3.4.2. Built-in annotations

The following built-in annotations are bundled with CIF:

- `controller:properties`
- `doc`
- `partial`
- `requirement:reachable`
- `state`

Controller properties annotations

Controller properties annotations add properties of a controller to a specification. For basic information on controller properties annotations, see the language tutorial. Here we discuss further details.

Constraints

Controller properties annotations (`@@controller:properties`) can only be added to the specification itself, not to elements of the specification.

The annotation has the following constraints, in addition to the general constraints that apply to all annotations:

- A specification may have at most one controller properties annotation.
- It is allowed for the annotation to not have any arguments, but in that case the annotation can

also just be removed.

- The arguments may be given in any order.
- The annotation supports the following arguments, and no other arguments are allowed:
  - `boundedResponse`:

    Indicates whether the specification has bounded response.

    Must have a boolean literal value. A `true` value indicates the specification has bounded response, and a `false` value indicates it does not have bounded response. If this argument is present and its value is `true`, then the `uncontrollablesBound` and `controllablesBound` arguments must also be present. If the argument is not present, or its value is `false`, these other two arguments must not be present.

  - `uncontrollablesBound`:

    Indicates the bound on the number of iterations for the event loop for uncontrollable events, if the specification has bounded response.

    Must have a non-negative integer literal value. This argument may only be present if the `boundedResponse` argument is also present, and its value is `true`.

  - `controllablesBound`:

    Indicates the bound on the number of iterations for the event loop for controllable events, if the specification has bounded response.

    Must have a non-negative integer literal value. This argument may only be present if the `boundedResponse` argument is also present, and its value is `true`.

  - `confluence`:

    Indicates whether the specification has confluence.

    Must have a boolean literal value. A `true` value indicates the specification has confluence, and a `false` value indicates it may not have confluence.

  - `finiteResponse`:

    Indicates whether the specification has finite response.

    Must have a boolean literal value. A `true` value indicates the specification has finite response, and a `false` value indicates it may not have finite response.

  - `nonBlockingUnderControl`:

    Indicates whether the specification is non-blocking under control.

    Must have a boolean literal value. A `true` value indicates the specification is non-blocking under control, and a `false` value indicates it is not non-blocking under control.

Documentation annotations

Documentation annotations can be used to add documentation to CIF elements. For basic information on documentation annotations, see the language tutorial. Here we discuss further details.

Constraints

Documentation annotations (`@doc`) can be added to all elements of CIF specifications that support annotations.

The annotation has the following constraints, in addition to the general constraints that apply to all annotations:

- The annotation must have at least one argument.
- All arguments must be unnamed arguments.
- All arguments must have a `string`-typed value.
- It must be possible for the argument values to be statically evaluated.
- Evaluation of the argument values must not result in an evaluation error.
- If multiple arguments are given, each argument becomes a separate line of text in the order that they are given.

Partial specification annotations

A partial specification annotation can be used to indicate that a CIF specification, such as a library, is meant to be included in other specifications. The annotation is used by the CIF type checker to disable certain types of warnings that may give a lot of false positives for partial specifications. For basic information on partial specification annotations, see the language tutorial. Here we discuss further details.

Constraints

Partial specification annotations (`@@partial`) can only be added to the specification itself, not to elements of the specification.

The annotation has the following constraints, in addition to the general constraints that apply to all annotations:

- A specification should have at most one partial specification annotation.
- The annotation must not have any arguments.

Scope

Unlike other annotations, a partial specification annotation applies only to the file in which it is specified. It is not inherited by importing models.

For instance, consider a library model that has a partial specification annotation. The library model is imported into a system model, which does not have a partial specification annotation. The library model has unused events, for which the warnings are suppressed. The system model is not a partial

model, and thus the warnings are reported, unless the events are used in the specification model. This prevents false positive warnings in the library model, while warnings about unused events still signal potential modeling issues in the system model.

Suppressed warnings

The following warnings are suppressed by the CIF type checker for partial specifications:

- Unused events.
- Channels without senders or receivers.
- Invariants that restrict events that are not in the synchronization, send or receive alphabet of the specification.

Reachability requirement annotations

A reachability requirement annotation can be used to specify a form of liveness requirements. A reachability requirement includes a goal predicate and may further restrict the source states, the states allowed on the path, and the maximum number of transitions to the goal. For the basic form without such restrictions, supervisory controller synthesis will ensure that in every state of the controlled system, either the goal predicate holds, or a state can be reached where it holds. For basic information on reachability requirement annotations, see the language tutorial. Here we discuss further details.

Constraints

Reachability requirement annotations (`@requirement:reachable`) can only be added to the following elements of the specification:

- To the root of the specification (using `@@requirement:reachable`).
- To components (automata and groups).
- To component definitions (automaton definitions and group definitions).
- To component instantiations (automaton instantiations and group instantiations).

The annotation has the following constraints, in addition to the general constraints that apply to all annotations:

- The annotation must have exactly one unnamed argument.
- The unnamed argument's value must be a predicate (a boolean-typed expression).
- Named and unnamed arguments may be specified in any order.
- Optional named arguments must be named `from`, `avoid`, `stay` or `bound`.
- The `from`, `avoid` and `stay` argument values must be predicates (boolean-typed expressions).
- The `bound` argument value must be a non-negative integer literal.
- The `avoid` and `stay` arguments may not both be specified.

Goal

The unnamed argument is the goal predicate, and it specifies the set of states that must remain reachable. Without further arguments, the goal must remain reachable from every state in the controlled system.

Conditional reachability with `from`

The `from` argument turns the requirement into a conditional reachability requirement and restricts the states for which it applies. If `from` is specified, only states that satisfy the source predicate must be able to reach the goal. States that do not satisfy the source predicate are not affected by the conditional reachability requirement.

Restricted reachability with `avoid` or `stay`

The `avoid` and `stay` arguments turn the requirement into a restricted reachability requirement. They restrict both the states for which the requirement applies and the states that may occur on the path to the goal.

With `avoid`, the path may only use states that do not satisfy the restriction predicate. The starting and goal states may not satisfy the restriction predicate either. States that already satisfy the restriction predicate are not affected by this reachability requirement.

With `stay`, the path may only use states that satisfy the restriction predicate. For states to which the requirement applies, this includes the starting state, every intermediate state, and the goal state. States that already do not satisfy the restriction predicate are not affected by this reachability requirement.

The `avoid` and `stay` arguments express the same path restriction in inverse forms: `avoid: pred` is equivalent to `stay: not pred`, and `stay: pred` is equivalent to `avoid: not pred`. The arguments may not both be specified, so use the form that best describes the intended behavior and avoids unnecessary logical negations.

The `from` argument can be combined with `avoid` or `stay` to further restrict the source states of a restricted reachability requirement. Then only source states inside the allowed path region must be able to reach the goal along a path in that region.

Bounded reachability with `bound`

The `bound` argument turns the annotation into a bounded reachability requirement. The goal must then be reachable using at most the specified number of transitions. For `bound: 0`, a state satisfies the requirement only if it already satisfies the goal predicate, unless it is not affected by the requirement due to a `from`, `avoid` or `stay` argument.

The `from` argument can also restrict bounded reachability to source states. This results in a conditional and bounded reachability requirement. If `from` is specified, only states that satisfy the source predicate must be able to reach the goal within the bound. States that already do not satisfy the source predicate are not affected by this conditional and bounded reachability requirement. If `bound` is specified without `from`, the source predicate is `true`, so there is no additional source-state restriction. The `avoid` and `stay` arguments can similarly be combined with `bound`, resulting in restricted and bounded reachability requirements. Conditional, restricted and bounded

reachability can all be combined in a single requirement.

For bounded reachability requirements, data-based synthesis debug output reports the configured bound, the minimum-bound status and the reason that the bounded computation stopped.

Scoping

If a reachability requirement annotation is specified on a component or component definition, then references to named elements are resolved from within the body of the component or component definition. For example:

```
input bool closed;

@requirement:reachable(closed)
plant bridge:
  controllable c_open, c_close;

  location open:
    initial;
    marked;
    edge c_close goto closed;

  location closed:
    edge c_open goto open;
end
```

In this case, the predicate refers to `closed`, and this is resolved to the location `closed` in the body of the `bridge` automaton (in the automaton's scope), rather than to the input variable named `closed` (in the specification's root scope).

State annotations

A state annotation adds state information to a location in an automaton. For basic information on state annotations, see the language tutorial. Here we discuss further details.

Constraints

State annotations (`@state`) can be added to the following elements in CIF specifications:

- Locations of automata.

The annotation has the following constraints, in addition to the general constraints that apply to all annotations:

- A single location may have multiple state annotations, if that location represents multiple states of a state space.
- Each argument of a state annotation represents either an automaton with its current location, or a variable with its current value.
- There can be any number of arguments (including no arguments), as models may have any

number of automata and variables.

- The arguments may be in any order.
- The name of the argument must be the absolute name of the corresponding automaton or variable.
- Argument values, and parts of argument values (in case of containers), must be literals of type `bool`, `int`, `real`, `string`, `tuple`, `list`, `set`, or `dict`. This implies that the values must be statically evaluable, and that evaluating an argument value must not result in an evaluation error.
- If at least one location in an automaton has at least one state annotation, all other locations in that same automaton must also have at least one state annotation, as all these locations should then represent a least one state from the state space.
- Different state annotations on the same or different locations of a single automaton must have the same number of arguments, the arguments must have the same names, and the values of arguments with matching names must have compatible types (ignoring ranges), as they should represent states from the same state space. The arguments don't have to be in the same order.

Most values can thus be represented as literals, but there are some exceptions. The following overview indicates how to represent locations and different types of values:

Location / value	**Represented as**
Location of an automaton	String literal with the non-escaped name of the location, or `"*"` for a nameless location.
Boolean value	`true` or `false` boolean literal.
Integer value	Integer literal, such as `0`, `1` or `1234`. Note that negative integer values are represented as a negation (unary operator) applied to a positive integer literal. And `-2147483648` is represented as `-(2147483647) - 1`.
Real value	Real literal, such as `0.0`, `1e5` or `0.5e-3`. Note that negative real values are represented as a negation (unary operator) applied to a positive real literal.
Enumeration value	String literal with the non-escaped name of the enumeration literal.
String value	String literal, such as `"some text"`.
Tuple value	Tuple literal, such as `(1, 2)` or `(true, (1, 2))`.
List value	List literal, such as `[1, 2]` or `[[1, 2], [3, 4]]`. Note that empty lists typically require a type hint in the form of a type cast, such as `<list[0] int>[]`.
Set value	Set literal, such as `{1, 2}` or `{{1, 2}, {3, 4}}`. Note that empty sets typically require a type hint in the form of a type cast, such as `<set int>{}`.
Dictionary value	Dictionary literal, such as `{1: 2, 3: 4}` or `{1: {2: 3}, 2: {4, 5}}`. Note that empty dictionaries typically require a type hint in the form of a type cast, such as `<dict(int:int)>{}`.
Function value	String literal with the non-escaped absolute name of the function, or `"*"` for functions that represent a default initial value of a variable with a function type.

Chapter 4. Tools

Below you can find information about the individual applications of the CIF toolset.

Applications

The CIF toolset consists of applications that cover a wide range of functionality. The available applications are listed below.

Specification

- CIF text editor

Analysis

- CIF information generator
- CIF relations collector

Supervisory controller synthesis

- Data-based supervisory controller synthesis
- CIF multi-level splitter
- Event-based synthesis toolset
- CIF to Supremica transformer

Simulation, validation, and verification

- CIF simulator
- CIF to mCRL2 transformer
- CIF to UPPAAL transformer
- Controller properties checker

Real-time testing, code generation, and implementation

- CIF code generator
- CIF PLC code generator (stable)
- CIF PLC code generator (deprecated)

Miscellaneous

- CIF to CIF transformer
- CIF merger
- CIF event disabler
- CIF explorer

- CIF to yEd transformer

Scripting

The CIF tools can be used on the command line, as well as in the Eclipse IDE. They can also be used in ToolDef scripts. The following additional information is available on scripting related to CIF and ToolDef scripts:

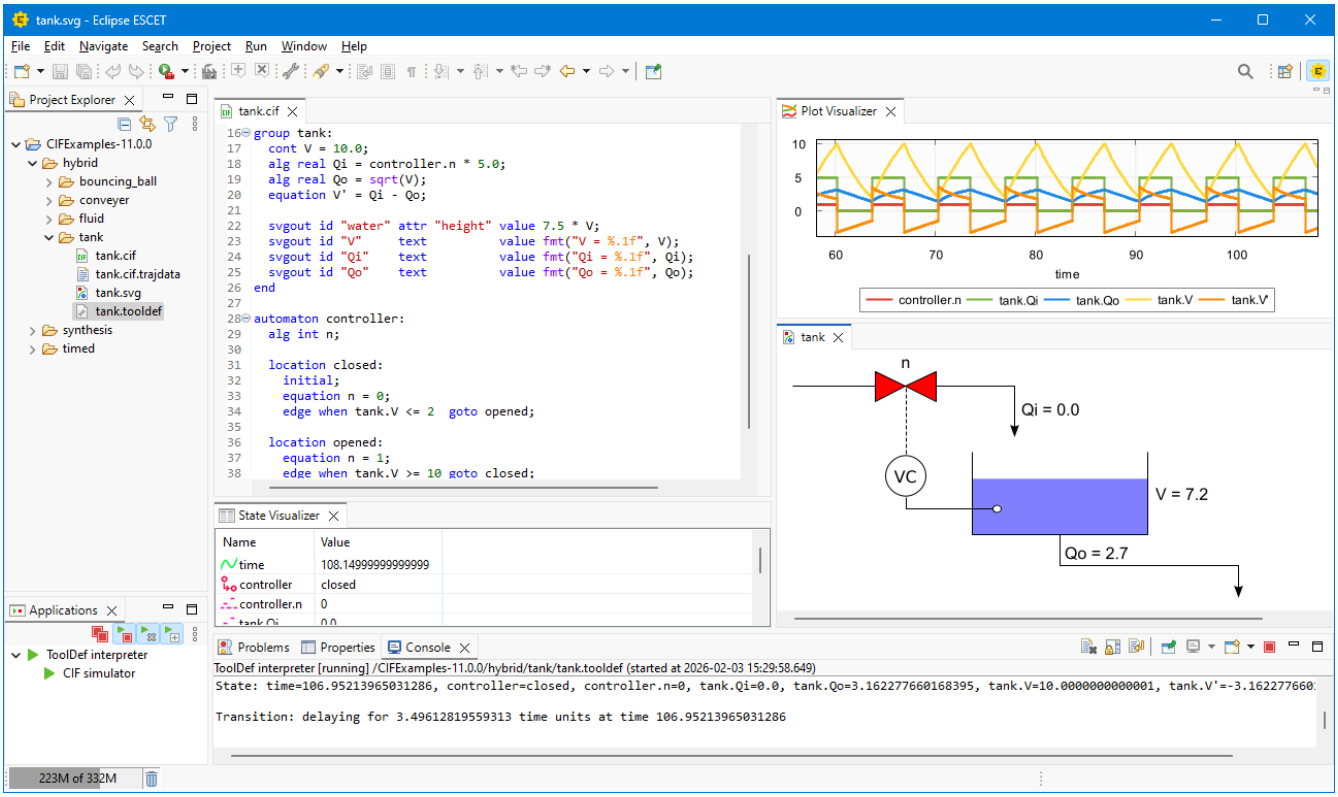
- Introduction to scripting
- Overview of scriptable tools

4.1. Specification tools

4.1.1. CIF text editor

Part of the CIF tooling is a textual editor for CIF specifications. This editor is part of the Eclipse ESCET IDE. Below is a list of some of the more notable features of this text editor:

- continuous background validation (integrated parsing and type checking), with error markings directly in the source
- comment spell checking
- code folding
- commands to comment/uncomment the (partially) selected line or lines
- block selection mode
- detection of external changes to files
- shows line numbers
- drag-and-drop editing
- optionally can show whitespace characters
- configurable tab key settings
- automatic removal of trailing whitespace from all lines on save
- automatic addition of a new line character at the end of the file on save, if none is present (includes smart detection of the platform dependent new line characters that are used in the file)



Normally, `.cif` files are opened with the CIF text editor. Large files are however opened with the default non-CIF text editor to avoid performance issues. You can open a file in an editor of your choosing by right clicking it and selecting **Open With** and then selecting the editor of your choosing, or choosing **Other...** to open a dialog to choose from a larger selection of editors.

4.2. Analysis tools

4.2.1. CIF information generator

The CIF information generator generates information about a CIF specification. The information to produce can be configured. The requested information is collected and written to a text file.

Starting the generator

The generator can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF analysis tools › Generate information...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF analysis tools › Generate information...**.
- Use the `cifinfogen` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifinfogen` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output information text file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.info.txt` file extension is added.
- *CIF information*: The information to produce. By default all information is produced. For more information, see the section on information to produce.

Supported specifications

The CIF information generator supports all CIF specifications.

The following CIF to CIF transformations are applied as preprocessing (in the given order), before the information is collected and written to the output file:

- Eliminate component definition/instantiation

Information to produce

The *CIF information* option can be used to specify which information to produce. By default, all information is produced. In the option dialog, simply select which information to produce. On the command line, specify comma-separated names of information to produce.

Below, the information that can be produced is explained in more detail. For each kind of information, after the name of the information, the command line name of that information is indicated between parentheses.

The following information can be produced:

- **Event-enablement conditions** (`event-enablement-conditions`)

  Generates per event declared in the CIF specification, a global condition that indicates when the event is enabled. The information that comprises the global condition is generally spread out over the CIF specification, for example in the guards of edges of various synchronizing automata. It can therefore be useful to see it all in one place. Inspecting the conditions allows to double check them and potentially spot modeling mistakes. Furthermore, if the information is produced for a synthesized supervisor, it can give insight into the behavior of the controlled system, taking into account the supervisor's guards.

  The enablement condition is derived from the automata that participate for the event, whether as a synchronizing automaton, send automaton, receive automaton or monitor automaton. Furthermore, all state/event exclusion invariants for the event are considered. The condition does not consider target state invariants. If the specification has any state invariants, a warning will be included in the output.

  For each declared event, its non-escaped absolute name is printed, followed by a colon and its condition predicate. As the `tau` event is not declared in the specification, nothing is printed for

`tau` events. The events are printed in sorted order.

- **Event participants** (`event-participants`)

  Generates per event declared in the CIF specification, an overview of what elements of the specification participate for the event. This information can be used to understand when the event is enabled. The participants can be spread out over the CIF specification, so the generated overview can be used to find the relevant elements of the specification. Inspecting the overview can potentially also reveal modeling mistakes.

  The information includes which automata participate for the event and in what role, namely as sender, receiver, synchronizer or monitor. It also includes the state/event exclusion invariants that relate to the event. State invariants are not included.

  For each declared event, its non-escaped absolute name is printed, with an indication of whether it is a channel or a regular event. As the `tau` event is not declared in the specification, nothing is printed for `tau` events. The events are printed in sorted order. For each event, per relevant category of participation, the participants are listed in sorted order. If for a relevant category there are no participants, this is explicitly indicated as well.

4.2.2. CIF relations collector

The CIF relations collector analyzes a CIF specification and produces three DMMs with the results. The DMMs describe found connections between plants and requirements. By automatically or manually studying the connections, the dependency structure between the CIF plants and requirements becomes more clear. The found structure can be exploited to split or modify a CIF specification in a useful way. For example, the multilevel tool uses it to create partial specifications.

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF analysis tools › Collect relations...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF analysis tools › Collect relations...**.
- Use the `cifrelations` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifrelations` command-line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative file system path to the input CIF file.
- *Plant-groups output file path*: The absolute or relative file system path to write the plant-groups DMM file. By default, or if no path is supplied, the application writes the plant groups DMM to a modified path of the *Input file* option. The path modification takes the file system path of the *Input file* option, then removes either a `.cif` or a `.cifx` suffix if it exists, and appends `.plant-groups.csv`. To skip writing the plant groups DMM, use `no` as file system path.
- *Requirement-groups output file path*: The absolute or relative file system path to write the requirement-groups DMM file. By default, or if no path is supplied, the application writes the requirement groups DMM to a modified path of the *Input file* option. The path modification takes the file system path of the *Input file* option, then removes either a `.cif` or a `.cifx` suffix if it exists, and appends `.requirement-groups.csv`. To skip writing the requirement groups DMM, use `no` as file system path.
- *Plants-requirements output file path*: The absolute or relative file system path to write the plants-requirements DMM file. By default, or if no path is supplied, the application writes the plants-requirements DMM to a modified path of the *Input file* option. The path modification takes the file system path of the *Input file* option, then removes either a `.cif` or a `.cifx` suffix if it exists, and appends `.plants-requirements.csv`. To skip writing the plant requirements DMM, use `no` as file system path.
- *RFC-4180 compliance*: Comma-separated value (CSV) files are defined in the RFC-4180 standard. The standard requires Windows style line delimiters, but using such files on other operating systems may cause problems. As a work-around, the CSV file writer can also use Unix style line delimiters, although that is not allowed in the RFC-4180 standard. However, as CSV files with Unix style line delimiters are accepted by all common operating systems including the Windows operating system, the default of this option is to ignore the Windows style line delimiter requirement of the standard and write Unix style line delimiters instead. If a fully compliant RFC-4180 CSV file is however required, enabling this option will cause CSV files to be written using Windows style line delimiters.

  Further details about the Windows and Unix style line delimiters, see the Wikipedia Newline article.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove annotations
- Eliminate component definition/instantiation
- Eliminate automaton `self` references
- Simplify values
- Remove I/O declarations

Supported specifications

The CIF relations collector tool has performed pre-processing before performing the checks below, which means that:

- Annotations are ignored, except for reachability requirement annotations (see also below).
- I/O declarations are ignored. A warning is printed if the specification contains a CIF/SVG input mapping.
- Component definitions and instantiations are allowed (they have been eliminated already).
- Automata references with `self` are allowed (they have been rewritten).

The CIF relations collector allows a subset of CIF specifications. The restrictions are:

- All automata must be either a plant or a requirement automaton. Kindless and supervisor automata are not allowed.
- Only requirement invariants are allowed. Kindless, plant and supervisor invariants are not allowed.
- Automata must have exactly one initial location, and it must be possible to determine this statically.
- The specification must have at least one input variable or one plant automaton. (An input variable counts as plant.)
- Discrete variables must have exactly one initial value, and it must be possible to determine this statically.
- The specification must have at least one requirement automaton or one requirement invariant.
- State requirement invariants are not allowed. Only state/event exclusion requirement invariants are allowed.
- Invariants in locations are not allowed. Only invariants in components are allowed.
- Reachability requirement annotations are not allowed.
- Tau events are not allowed.
- Continuous variables are not allowed.
- Equations are not allowed.
- Component types are not allowed.

Output of the CIF relations collector

The CIF relations collector analyzes the plants and requirements of a CIF specification, and can produce three Domain Mapping Matrix (DMM) files that contain the results. A DMM is a variant of a Design Structure Matrix (DSM), where connections between two different domains are expressed. In a DMM, each row represents a value of the first domain, and each column represents a value of the second domain. Each entry `(row, column)` in the matrix has the value `1` if the `row` value of the first domain is related to the `column` value of the second domain, otherwise the entry is `0`.

The following DMM files can be produced as results:

- The 'plant-groups' DMM describes how elementary CIF plant groups are combined into larger plant groups.
- The 'requirement-groups' DMM describes how elementary CIF requirement groups are combined into larger requirement groups.
- The 'plants-requirements' DMM describes how requirement groups relate to plant groups.

Elementary groups

Each plant automaton and each input variable is considered to be a plant group. Each requirement automaton and each requirement state/event exclusion invariant in a component is considered to be a requirement group.

Merging plant groups and merging requirement groups

Each plant group and each requirement group should be self-contained. This goal is achieved by repeatedly merging two plant groups into one (larger) group, or merging two requirement groups into one (larger) group.

Two plant groups should be merged if and only if at least one of the following holds:

- One plant group owns and the other plant group accesses a same location or discrete variable.
- Both plant groups access a same location or discrete variable.
- Both plant groups access a same input variable or event.

Two requirement groups should be merged if and only if at least one of the following holds:

- One requirement group owns and the other requirement group accesses a same location or discrete variable.
- Both requirement groups access an event that is not accessed by any plant group.

Ownership of a discrete variable by a group means that the discrete variable is in a plant automaton of that group. Ownership of a location by a group means that the location is in a plant automaton of that group. Accessing means that the group refers to a CIF element by its name. For example, a requirement like `requirement start needs aut.ready` accesses the `start` event and the `ready` location of automaton `aut`.

Relating plant groups with requirement groups

A plant group is related to a requirement group if and only if at least one of the following holds:

- One group owns a location and the other group accesses that location.
- One group owns a discrete variable and the other group accesses that variable.
- Both groups access the same input variable.
- Both groups access the same event.

Note that having a requirement group that is not related to any of the plant groups is allowed by

these rules. As this may indicate a modeling mistake, the tool warns about such requirement groups.

Further reading

The definition of plant groups, requirement groups, and plants-requirements relations is based on [Goorden et al. (2020)], but with extensions.

- The CIF relations collector is more aligned with features available in CIF.
- The CIF relations collector features requirement groups to handle requirements that are only indirectly related to plants, which is new with respect to the paper.

References

- [Goorden et al. (2020)] Martijn Goorden, Joanna van de Mortel-Fronczak, Michel Reniers, Wan Fokkink and Jacobus Rooda, "Structuring Multilevel Discrete-Event Systems With Dependence Structure Matrices", IEEE Transactions on Automatic Control, volume 65, issue 4, pages 1625-1639, 2020, doi:10.1109/TAC.2019.2928119

4.3. Supervisory controller synthesis tools

4.3.1. Data-based supervisory controller synthesis

The data-based supervisory controller synthesis tool performs data-based supervisory controller synthesis, or simply data-based synthesis. It can be used to synthesize a supervisor for an untimed CIF specification, with data (e.g. discrete variables). Synthesis is an essential part of the synthesis-based engineering approach to develop supervisory controllers.

For a CIF specification with plants and requirements, the tool computes a supervisor. The supervisor restricts the plants in such a way that the resulting controlled system satisfies the following properties:

- The controlled system is safe. That is, all reachable states in the controlled system satisfy the requirements.
- The controlled system is controllable. That is, for all reachable states in the controlled system, the uncontrollable events that are enabled in the same state in the uncontrolled system are still possible in the controlled system. In other words, uncontrollable events are not restricted.
- The controlled system is non-blocking. That is, it is possible to reach a marked state from all reachable states in the controlled system.
- The controlled system is maximally permissive (or simply maximal). That is, the controlled system permits all safe, controllable, and non-blocking behaviors.

Note that deadlock is not prevented for marked states, nor for states that satisfy reachability requirements.

The synthesis algorithm is based on [Ouedraogo et al. (2011)] and explained in quite some detail in [Hendriks et al. (2025)].

The following additional information is available:

- Starting the data-based synthesis tool
- Data-based synthesis options
- Specifications supported by data-based synthesis
- Early problem detection
- Resulting supervisor
- Predicate simplification
- Forward reachability
- Fixed-point computations order
- Input variables
- Initialization
- Performance
- Edge granularity
- Edge order
- BDD variable order
- BDD operation cache
- State requirement invariants
- Debug output
- Statistics
- References

Starting the data-based synthesis tool

The data-based synthesis tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Apply data-based synthesis...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Apply data-based synthesis...**.
- Use the `cifdatasynth` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifdatasynth` command line tool.

Data-based synthesis options

Besides the general application options, the data-based synthesis tool has several other options.

The following options are part of the *Synthesis* category:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output CIF file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.ctrlsys.cif` file extension is added.
- *Supervisor name*: The name of the resulting supervisor automaton. If not specified, it defaults to `sup`. For more information, see the section about the resulting supervisor.
- *Supervisor namespace*: The namespace of the resulting supervisor. If not specified, it defaults to the empty namespace. For more information, see the section about namespaces.
- *Forward reachability*: Whether to perform forward reachability during synthesis, or omit it. Is disabled by default. For more information, see the section about forward reachability.
- *Fixed-point computations order*: The order in which the different forward/backward reachability fixed-point computations are to be performed during synthesis. This may impact the performance of synthesis. For more information, see the section about fixed-point computations order.
- *Exploration strategy*: The exploration strategy to use for symbolic reachability computations. This strategy determines in which order the edges are to be considered. This may impact the performance of synthesis. For more information, see the section about edge order.
- *Edge granularity*: Specify the granularity of edges to use during synthesis. For more information, see the section about edge granularity.
- *Edge order*: This option is no longer supported. Use the *Edge order for backward reachability* and *Edge order for forward reachability* options instead.
- *Edge order for backward reachability*: Synthesis involves many reachability computations. Some of them involve following edges backward to find reachable states. The order in which the edges are considered for such backward reachability computations is determined by this option. For more information, see the section about edge order.
- *Edge order for forward reachability*: Synthesis involves many reachability computations. Some of them involve following edges forward to find reachable states. The order in which the edges are considered for such forward reachability computations is determined by this option. For more information, see the section about edge order.
- *Edge order duplicate events*: Specify whether duplicate events are allowed for custom edge orders specified using the *Edge order for backward reachability* and *Edge order for forward reachability* options. By default, duplicate events are disallowed. For more information, see the section about edge order.
- *Edge workset algorithm*: This option is no longer supported. Use the *Exploration strategy* option instead.
- *State requirement invariant enforcement*: Specify how state requirement invariants are enforced during synthesis. By default, they are all enforced via the controlled behavior. But, this can also be decided per edge, enforcing them via edge guards for edges with controllable events, and enforcing them via the controlled behavior for edges with uncontrollable events. For more information, see the section on state requirement invariants.

- *Statistics*: The kinds of statistics to print. By default, no statistics are printed. For more information, see the section about statistics.
- *Continuous performance statistics file*: The absolute or relative local file system path to the continuous performance statistics output file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.stats.txt` file extension is added. For more information, see the section about statistics.
- *Event warning*: Whether to warn for events that are never enabled in the input specification or never enabled in the controlled system. Is enabled by default. Disabling this warning may increase the performance of synthesis. For more information, see the section about early problem detection.
- *Plants referencing requirements warnings*: Whether to warn for plants that reference requirement state. Is enabled by default. Disabling this warning may increase the performance of synthesis. For more information, see the section about early problem detection.

Internally during synthesis, predicates are represented using Binary Decision Diagrams (BDDs). There are various options that can influence the use of BDDs. The following options are part of the *BDD* sub-category of the *Synthesis* category:

- *BDD output mode*: This option can be used to control how the BDDs are converted to CIF for the output of synthesis. For more information, see the section about BDD representations of the resulting supervisor.
- *BDD output name prefix*: The prefix to use for BDD related names in the output. Only has an effect if the *BDD output mode* option is set to represent the internal BDD nodes directly in CIF. The default prefix is `bdd`. For more information, see the section about BDD representations of the resulting supervisor.
- *BDD initial variable ordering*: CIF variables and automata are represented using one or more boolean variables. The initial order of the boolean variables is determined by this option. For more information, see the section about BDD variable orders and in particular the section about initial variable ordering.
- *BDD hyper-edge creation algorithm*: CIF variables and automata are represented using one or more boolean variables. The initial order can be improved and optimized by various variable ordering algorithms. This option controls how hyper-edges are created, which are used by the variable ordering algorithms. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.
- *BDD hyper-edge graph creation algorithm*: CIF variables and automata are represented using one or more boolean variables. The initial order can be improved and optimized by various variable ordering algorithms, which may rely on graph representations of variable relations. This option controls how the graphs are created. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.
- *BDD DCSH variable ordering algorithm*: CIF variables and automata are represented using one or more boolean variables. The initial order of the boolean variables can be improved by enabling this option. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.
- *BDD FORCE variable ordering algorithm*: CIF variables and automata are represented using one

or more boolean variables. The initial order of the boolean variables can be improved by enabling this option. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.

- *BDD sliding window variable ordering algorithm*: CIF variables and automata are represented using one or more boolean variables. The initial order of the boolean variables can be improved by enabling this option. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.
- *BDD sliding window size*: The maximum length of the window to use for the BDD sliding window variable ordering algorithm. This option only has effect if the *BDD sliding window variable ordering algorithm* option is enabled. The size must be an integer number in the range [1 .. 12]. The default size is 4. For more information, see the section about BDD variable orders and in particular the section about automatic variable ordering.
- *BDD advanced variable ordering*: CIF variables and automata are represented using one or more boolean variables. The variable ordering can be configured most flexibly using this option. For more information, see the section about BDD variable orders and in particular the section about advanced variable ordering configuration.
- *BDD predicate simplify*: Several BDD predicates may be simplified under the assumption of other predicates, resulting in smaller/simpler output. This may decrease the size of the resulting controller, and may give more insight. For more information, see the section about predicate simplification.
- *BDD library initial node table size*: The BDD library that is used maintains an internal node table. This option can be used to set the initial size of that node table. The size will automatically be increased during synthesis, if necessary. Increasing the initial size can increase performance for large systems, as it will not be necessary to repeatedly increase the size of the node table. However, a larger node table requires more memory, and can lead to the node table no longer fitting within CPU caches, degrading performance. The default is `100000` nodes. The initial node table size must be in the range [2 .. 2^{31} - 1]. For more information, see the section about statistics.
- *BDD library operation cache size*: The BDD library that is used maintains several internal operation caches. This option can be used to set the fixed size of these caches. The operation cache size must be in the range [2 .. 2^{31} - 1]. By default, this option is disabled (value `off` on the command line), and the *BDD library operation cache ratio* option is used instead. For more information, see the section about BDD operation caches.
- *BDD library operation cache ratio*: The BDD library that is used maintains several internal operation caches. This option can be used to set the ratio of the size of the operation caches of the BDD library to the size of the node table of the BDD library. For instance, a ratio of `0.1` means that the sizes of the operation caches are 10% of the size of the node table. The operation cache ratio must be in the range [0.01 .. 1000]. The default ratio is 1.0. This option has no effect if the *BDD library operation cache size* option is enabled. For more information, see the section about BDD operation caches.
- *BDD debug max nodes*: Internally, predicates are represented using Binary Decision Diagrams (BDDs). This option control the maximum number of BDD nodes for which to convert a BDD to a readable CNF/DNF representation for the debug output. The default is `10` nodes. The maximum must be in the range [1 .. 2^{31} - 1]. The option can be set to have an infinite maximum (no

maximum), using option value `inf`. For more information, see the section about debug output.

- *BDD debug max paths*: Internally, predicates are represented using Binary Decision Diagrams (BDDs). This option control the maximum number of BDD true paths for which to convert a BDD to a readable CNF/DNF representation for the debug output. The default is `10` paths. The maximum must be non-negative. The option can be set to have an infinite maximum (no maximum), using option value `inf`. For more information, see the section about debug output.

Specifications supported by data-based synthesis

The data-based supervisory controller synthesis tool supports a subset of CIF specifications. The following restrictions apply:

- Only `plant` and `requirement` automata are supported. Automata with a `supervisor` kind, as well as kindless/regular automata (without a supervisory kind) are not supported.
- Specifications without plant automata are not supported.
- Events not declared as controllable or uncontrollable are not supported. This includes the `tau` event, both explicitly used on edges, as well as implicitly for edges without explicitly mentioned events.
- The use of channels (events with data types) in requirements is not supported. That is, requirements that send or receive (with or without data) are not supported.
- Only `plant` and `requirement` invariants are supported. Invariants with a `supervisor` kind, as well as kindless/regular invariants (without a supervisory kind) are not supported.
- Continuous variables are not supported.
- Discrete and input variables are supported, but only if they have a supported type:
  - Boolean type.
  - Ranged integer type (e.g. `int[0..5]`). Ranges that allow more than 2,147,483,647 (2^31 - 1) possible values are not supported.
  - Enumeration type.
  - Tuple type, with supported types for the tuple's fields.
- Discrete variables must have supported initial values. If explicit initial values are given, they must be supported predicates (for boolean variables) or supported expressions as described below (for all other variables).
- Constants, algebraic variables and algebraic parameters of components are supported as long as their values are supported in the context where they are used (as described below).
- Automata with non-determinism for controllable events are not supported. That is, automata that have locations with multiple outgoing edges for the same controllable event, with overlapping guards (e.g. `x > 1` and `x < 4`), are not supported. Note that this check is performed on the linearized guards, and may therefore lead to false positives, as the check is an over-approximation and guard overlap may be detected for unreachable states.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.

- Annotations are ignored, except for reachability requirement annotations.

The following restrictions apply to user-defined functions:

- External user-defined functions are not supported.
- Internal user-defined functions with recursive calls are not supported.
- Internal user-defined functions with `while` statements are not supported.
- Internal user-defined functions with assignments that may result in runtime out-of-range errors are not supported.
- Internal user-defined functions may only be used in function calls. All other uses are not supported, including using them as data, such as storing them in variables, returning them from functions, and so on.
- Internal user-defined functions may only use supported data types (same as discrete and input variables) as their return and parameter types, and as the types of their local variables.

Only limited forms of predicates (for markers, invariants, initialization, guards, initial values of boolean variables, right hand sides of assignments, etc) are supported. The supported predicates are:

- Boolean literals (`true`, `false`).
- Discrete and input boolean variables (`x`, for `x` a discrete or input variable with a boolean type).
- Algebraic boolean variables with supported predicates as their values (`x`, for `x` an algebraic variable with a boolean type).
- Boolean constants (`x`, for `x` a constant with a boolean type).
- Locations (`aut.loc`, for `aut` and automaton and `loc` a location of that automaton).
- The unary inverse operator (`not`) on a supported predicate.
- The binary logical conjunction (`and`) on two supported predicates.
- The binary logical disjunction (`or`) on two supported predicates.
- The binary logical implication (`=>`) on two supported predicates.
- The binary logical bi-conditional (`<=>`) on two supported predicates.
- The binary equality comparison (`=`) on two supported predicates, on two supported integer or enumeration expressions, or on two supported tuple expressions.
- The binary inequality comparison (`!=`) on two supported predicates, on two supported integer or enumeration expressions, or on two supported tuple expressions.
- The binary less than comparison (`<`) on two supported integer expressions.
- The binary less than or equal comparison (`<=`) on two supported integer expressions.
- The binary greater than comparison (`>`) on two supported integer expressions.
- The binary greater than or equal comparison (`>=`) on two supported integer expressions.
- Conditional expressions (`if` expressions) with supported guard and resulting value predicates.
- `Switch` expressions with supported control value and resulting value predicates.

- Function calls to internal user-defined functions with supported arguments that return a boolean value.
- Projections on tuples that result in a boolean value.
- Received values (`?`) with a boolean type.
- Any other valid CIF predicate (computation) that results in a boolean value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

  Here are some examples of computations that can be statically evaluated:

  - `{1} in {1, 2}` (result is `true`)
  - `[false, true][0]` (result is `false`)
  - `c - 1`, for a constant `c` with value `4` (result is `3`)

  Here are some examples of computations that can *not* be statically evaluated:

  - `v = true` for `v` a discrete variable. The computation results in different values for different values of `v`.
  - `{v} in {1, 2}` for `v` a discrete variable. The computation results in different values for different values of `v`.

Only limited forms of integer and enumeration expressions (for binary comparisons, initial values of variables, right hand sides of assignments, etc) are supported. The supported expressions are:

- An integer literal/value.
- An enumeration literal/value.
- Discrete and input integer/enumeration variables (`x`, for `x` a discrete or input variable with an integer or enumeration type).
- Algebraic integer/enumeration variables with supported expressions as their values (`x`, for `x` an algebraic variable with an integer or enumeration type).
- Integer/enumeration constants (`x`, for `x` a constant with an integer or enumeration type).
- `+i` and `-i` for `i` a supported integer expression.
- `i + j`, `i - j`, and `i * j` for `i` and `j` supported integer expressions.
- `i div j` and `i mod j` for `i` a supported integer expressions, and `j` a positive integer value, or a computation that results in a positive integer value, as long as the computation is not too complex to be performed statically. That is, `j` must essentially be constant.
- `abs(i)` and `sign(i)` for `i` a supported integer expression.
- `max(i, j)` and `min(i, j)` for `i` and `j` supported integer expressions.
- Conditional expressions (`if` expressions) with supported guard predicates and supported resulting values.
- `Switch` expressions with supported control values and supported resulting values.
- Function calls to internal user-defined functions with supported arguments that return an

integer or enumeration value.

- Projections on tuples that result in an integer or enumeration value.
- Received values (`?`) with an integer or enumeration type.
- Any other valid CIF expression (computation) that results in an integer value or an enumeration value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

  Here are some examples of computations that can be statically evaluated:

  - `2 * 5` (result is `10`)
  - `floor(3.14)` (result is `3`)
  - `c + 1`, for a constant `c` with value `2` (result is `3`)

  Here are some examples of computations that can *not* be statically evaluated:

  - `v - 1`, for `v` a discrete variable. The computation results in different values for different values of `v`.
  - `ceil(sqrt(-1.0))`, since `sqrt` of a negative number results in a runtime error upon evaluation.

Only limited forms of tuple expressions (for binary comparisons, initial values of variables, right hand sides of assignments, etc) are supported. The supported expressions are:

- A tuple literal/value.
- Discrete and input tuple-typed variables (`x`, for `x` a discrete or input variable with a tuple type).
- Algebraic tuple-typed variables with supported expressions as their values (`x`, for `x` an algebraic variable with a tuple type).
- Tuple-typed constants (`x`, for `x` a constant with a tuple type).
- Conditional expressions (`if` expressions) with supported guard predicates and supported resulting tuple-typed values.
- `Switch` expressions with supported control values and supported resulting tuple-typed values.
- Function calls to internal user-defined functions with supported arguments that return a tuple value.
- Projections on tuples that result in a tuple value.
- Received values (`?`) with a tuple type.
- Any other valid CIF expression (computation) that results in a tuple value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to

increase the subset of CIF specifications that can be synthesized:

- Remove I/O declarations
- Eliminate component definition/instantiation

Additionally, the CIF specification is converted to an internal representation on which the synthesis is performed. This conversion also applies linearization ('product with merge' variant) to the edges. Predicates are represented internally using Binary Decision Diagrams (BDDs).

Supported requirements

Four types of requirements are supported: state invariants, state/event exclusion invariants, requirement automata and reachability requirements. For state invariants and state/event exclusion invariants, both named and nameless variants are supported.

State invariants are global conditions over the values of variables (and locations of automata) that must always hold. Such requirements are sometimes also called mutual state exclusions. Here are some examples:

```
requirement invariant x != 0 and not p.b;
requirement invariant x > 5;
requirement R1: invariant not(x = 1 and y = 1) or q.x = a;

requirement (x = 1 and y = 1) or (x = 2 and y = 2);
requirement (3 <= x and x < = 5) or (2 <= y and y <= 7);
requirement x = 1 => y > 2;
```

State/event exclusion invariants or simply state/event exclusions are additional conditions under which transitions may take place for certain events. Here are some examples:

```
requirement invariant buffer.c_add    needs buffer.count < 5;
requirement invariant buffer.c_remove needs buffer.count > 0;
requirement invariant button.on = 1 disables lamp.c_turn_on;
requirement invariant R3: buffer.c_remove needs buffer.count > 0;

requirement {lamp.c_turn_on, motor.c_turn_on} needs button.Off;
requirement p.x = 3 and p.y > 7 disables p.u_something;
```

Requirement automata are simply automata marked as `requirement`. They usually introduce additional state by using multiple locations or a variable. The additional state is used to be able to express the requirement. One common example is a counter. For instance, consider the following requirement, which prevents more than three products being added to a buffer:

```
requirement automaton counter:
  disc int[0..5] count = 0;
```

```
  requirement invariant count <= 3;

  location:
    initial;
    marked;

    edge buffer.c_add do count := count + 1;
end
```

Another common example is a requirement that introduces ordering. For instance, consider the following requirement, which states that `motor1` must always be turned on before `motor2` is turned on, and they must always be turned off in the opposite order:

```
requirement automaton order:
  location on1:
    initial;
    marked;
    edge motor1.c_on goto on2;

  location on2:
    edge motor2.c_on goto off2;

  location off2:
    edge motor2.c_off goto off1;

  location off1:
    edge motor1.c_off goto on1;
end
```

Reachability requirements can be used to specify liveness conditions, in addition to marker predicates.

Besides the explicit requirements, synthesis also prevents runtime errors. This includes enforcing that integer (parts of) variables stay within their range of allowed values. This is essentially an implicit requirement. For instance, for a CIF specification with a variable `x` of type `int[0..5]` and a variable `y` of type `int[1..3]`, requirement invariant `0 <= x and x <= 5 and 1 <= y and y <= 3` is implicitly added and enforced by the synthesis algorithm. In the resulting controlled system, no runtime errors due to (parts of) variables being assigned values outside their domain (integer value range) occur.

Early problem detection

The data-based synthesis tool checks the specification for common issues, for early detection of problems that will lead to an empty supervisor. If such a problem is detected, a warning is printed to the console. Among others, checks are included for no initial states/variables, no marked states, and no states due to the state requirement invariants. The tool also checks whether a reachability

requirement, by itself or in combination with plant and/or requirement state invariants, leaves no possible states in the controlled system. To skip printing these empty supervisor warnings, configure the *Output mode* option to only print errors (*General* category).

The tool also checks whether there are events that are never enabled in the input specification. If such a problem is detected, a warning is printed to the console. Among others, checks are included for events that are forbidden by automaton guards, event/state exclusion plant and requirement invariants, and state plant invariants. To skip these event checks, disable the *Event warning* option (*Synthesis* category).

The tool furthermore checks for plant invariants or plant automata that reference requirement state. If such a problem is detected, a warning is printed to the console. Among others, references in guards, updates, invariant predicates, initialization predicates, marker predicates and initial values for discrete variables are detected. To skip these reference checks, disable the *Plants referencing requirements warnings* option (*Synthesis* category).

Resulting supervisor

If the supervisor has to restrict so much of the behavior of the uncontrolled system that no initial state remains, the controlled system becomes empty. The data-based synthesis algorithm then ends with an *empty supervisor* error, and no output CIF file is created.

If an initial state remains after synthesis, an output CIF file is created. The contents is the controlled system. The controlled system is obtained by taking the input specification, and modifying it. The requirement automata are changed to supervisor automata. Some or all of the requirement invariants may be removed, depending on the simplifications that are applied. The remaining requirement invariants are changed to supervisor invariants. The reachability requirement annotations are preserved. An additional external supervisor automaton is added. Also, depending on the simplifications that are applied, the requirement automata may serve as monitors or *observers* for the external supervisor, or may actually impose the requirement restrictions. An external supervisor is a `supervisor` automaton that adds restrictions to the uncontrolled system (the plants), and potentially the requirement automata, depending on the simplifications that are applied. The supervisor uses the same events as the plants, and refers to plant and requirement locations and variables in its conditions.

By default, the resulting external supervisor automaton is put in the empty namespace, at the top level of the resulting specification. That is, the supervisor automaton is not put in any groups. See the namespace section for more information.

By default, the added supervisor automaton is named `sup`. Using the *Supervisor name* option (see the options section), it is possible to specify a different name. Custom supervisor automaton names must be valid CIF identifiers, i.e. they may consist of letters, digits, and underscores (_), but may not start with a digit. If the resulting supervisor automaton has a name that conflicts with an existing declaration, it is automatically renamed to have a non-conflicting name. In such cases, a warning is printed to the console to inform the user.

The resulting supervisor has exactly one self loop edge for each of the controllable events in the alphabet of the controlled system (which is equal to the alphabet of the uncontrolled system). These

self loops represent the possible conditions under which the supervisor allows the events to occur in the controlled system. The exact predicates may vary, depending on the simplifications that are applied.

If there are controllable events that are never enabled in the controlled system, a warning is printed to the console. Enabling forward reachability ensures that warnings are also printed for events that are only enabled in the unreachable part of the statespace. Disabling forward reachability may lead to false negatives, i.e. such cases may not be reported. To skip checking the controllable events, disable the *Event warning* option (*Synthesis* category).

The resulting supervisor may have an initialization predicate that restricts the states in which the system may be initialized (may start), on top of the initialization constraints already present in the uncontrolled system. For more information on this initialization predicate, see the section about initialization.

The added supervisor automaton and all its elements do not have any annotations. The output specification never has a controller properties annotation, even if the input specification has one.

Namespace

As indicated above, by default the resulting supervisor automaton is put in the empty namespace, at the top level of the resulting specification. That is, the supervisor automaton is not put in any groups.

It is possible to add a namespace to the entire resulting specification. That is, to put groups around the original plants and requirements, the added supervisor automaton, etc. A namespace can be added using the *Supervisor namespace* option (see the options section). By default, no additional namespace is added.

By adding a namespace around the entire resulting specification, the synthesis result can be easily merged with for instance a simulation model. The added namespace ensures that there are no naming conflicts between the plants of the simulation model and the similarly original plants. The events are not put in the new namespace, but are instead kept in their original place, wrapped in groups as necessary to keep their original identities (absolute names). This ensures that it remains possible to connect (merge) the events of the synthesis output with the events of the simulation model.

The namespace specified using the option, must consist of one or more valid CIF identifiers, separated by dots (`.`). Valid CIF identifiers consist of one or more letters, digits, and underscores (_), but may not start with a digit. As an example, consider namespace `a.b`. A group `b` is wrapped around the entire synthesis result, and a group `a` is wrapped around group `b`. Group `a` is then the new synthesis result.

If a part of the namespace has the same name as an event that remains in its original place, this leads to a conflict, and synthesis fails. If the namespace does not conflict, but is non-empty (it contains an event or it contains a group that contains an event), synthesis also fails.

BDD representation in CIF

Internally, predicates are represented using Binary Decision Diagrams (BDDs). The supervisor that

is the output of synthesis, contains several predicates as well. For instance, it has self loops with guard predicates, and it may have an initialization predicate. The predicates represented as BDDs need to be represented as CIF predicates. There are multiple approaches to do this, and the *BDD output mode* option (see the options section), can be used to configure the approach to use.

The first approach, which is also the default approach (named `normal`), is to use either Conjunctive Normal Form (CNF) or Disjunctive Normal Form (DNF) predicates. Both a CNF predicate and a DNF predicate are created and the smallest one is used. If they have the same size, the CNF predicate is used. This approach has as benefit that for relatively small predicates (usually for small systems), the CIF predicates are often intuitive, and can easily understood. The downside is that for larger systems, the CIF predicates often grow exponentially in size.

The second approach (named `cnf`) always uses the CNF representation. The third approach (named `dnf`) always uses the DNF representation. Selecting one representation, when the other blows up exponentially, can prevent long running times and out-of-memory errors. But, knowing which one to select can be challenging, as this typically depends on the model being synthesized.

The fourth approached (named `nodes`), is to represent the internal BDD nodes directly in CIF. The BDD is then more or less coded directly in CIF, using some constants and algebraic variables, and is evaluated using a BDD evaluation function. The benefit is that for larger systems, this representation remains relatively small, and at the very least doesn't blow up nearly as much as the CNF and DNF representations. The downside to this approach, is that it leads to a supervisor that can not be easily understood by inspecting it. For this approach, several objects are created in the top level scope of the CIF specification. The names of these declarations all share a common prefix. The default prefix is `bdd`, but it can be changed using the *BDD output name prefix* option (see the options section). No existing declarations, whose names start with that prefix, should be present in the top level scope of the specification.

Predicate simplification

The data-based synthesis algorithm computes various predicates, such as the conditions under which the controllable events may take place in the controlled system, and the initialization predicate of the controlled system. These predicates are included in the supervisor that results from synthesis.

However, if the controlled system imposes the exact same restrictions as the uncontrolled system, there is no need to list the full conditions in the supervisor, as the plants already define that behavior. The supervisor imposes no additional restrictions with respect to the plants, and it suffices to use `true` as condition for the supervisor to make that explicit.

There are several predicates in the synthesis result that can be simplified under the assumption of conditions that are already present in the input specification. In some cases this leads to smaller/simpler supervisor representations. In other cases it gives insight, indicating that the supervisor does not impose any additional restrictions. The following simplifications are available:

Option value	Default	Predicate	May be simplified assuming
`guards-plants`	yes	Supervisor guards of controllable events	Plant guards, for the matching events
`guards-req-auts`	yes	Supervisor guards of controllable events	State/event exclusion requirement invariants derived from the requirement automata, for the matching events
`guards-se-excl-plant-invs`	yes	Supervisor guards of controllable events	State/event exclusion plant invariants from the input specification, for the matching events
`guards-se-excl-req-invs`	yes	Supervisor guards of controllable events	State/event exclusion requirement invariants from the input specification, for the matching events
`guards-state-plant-invs`	yes	Supervisor guards of controllable events	State plant invariants from the input specification
`guards-state-req-invs`	yes	Supervisor guards of controllable events	State requirement invariants from the input specification (includes the range requirement invariants added by the synthesis algorithm)
`guards-ctrl-beh`	yes	Supervisor guards of controllable events	Controlled behavior as computed by synthesis
`initial-unctrl`	yes	Initialization predicate of the controlled system	Initialization predicate of the uncontrolled system
`initial-state-plant-invs`	yes	Initialization predicate of the controlled system	State plant invariants from the input specification

Which simplifications should be performed, can be specified using the *BDD predicate simplify* option (see the options section).

The table above lists in the first column, the option values to use for each of the simplifications, on the command line. The names given in the first column should be combined using commas, and used as option value. The simplifications that are specified using the option replace the default simplifications (see the second column of the table). However, it is also possible to specify additions and removals relative to the default simplifications, by prefixing simplifications (from the first column) with a `+` or `-` respectively. Replacements (no prefix) may not be combined with additions/removals (`+` or `-` prefix). Specifying a simplification twice leads to a warning being printed to the console. Adding a simplification that is already present or removing a simplification that is not present, also leads to a warning being printed.

In the option dialog, each of the simplifications can be enabled or disabled using a checkbox.

The second column indicates for each simplification whether it is enabled by default. By default, all simplifications are enabled. The third column indicates the predicate in the synthesis result that can be simplified. The fourth column indicates under the assumption of which predicate the simplification is applied.

The simplification algorithm is not perfect, and may not simplify the predicates as much as could potentially be possible.

When simplifying with respect to state requirement invariants, the supervisor no longer enforces those requirements, as they are assumed to already hold. As such, the simplification prevents such invariants from being removed from the resulting CIF specification. This applies to some of the other simplifications as well. For instance, the simplification over state/event exclusion requirement invariants leads to them being part of the output as well. This may affect whether other tools can handle the resulting supervisor model as input, depending on what kind of features they support. In particular, for code generation, simplification of the guards with respect to the state requirement invariants may need to be disabled.

Forward reachability

Data-based synthesis essentially works by computations that involve predicates that partition the entire state space into states that satisfy a property or don't satisfy a property. For instance, a marker predicate may indicate which states of the state space are marked. All other states are thus not marked.

The main computations performed during synthesis involve reachability computations. For instance, from which states is it possible to reach a marker state? This form of reachability is called *backward* reachability, as it starts with some target (e.g. marked states), and goes backwards to find all states from which the target can be reached. Backward reachability can lead to states that could never be reached from an initial state, even in the uncontrolled system. This leads to two separate issues.

The first issue is about unintuitive resulting supervisor guards. The resulting supervisor forbids certain transitions, by restricting controllable events. It among others forbids transitions that end up in states from which no marked state can be reached. However, if those forbidden states can never be reached from an initial state, there is no reason to restrict the controllable events in such cases. The guards of the resulting supervisor then appear to restrict the controllable events, while in fact the guard doesn't impose a restriction for the controlled system. The supervisor simply doesn't have the necessary information to know this.

The second issue is about performance. Expanding unreachable states during backward reachability takes time and costs memory, while it has no useful effect on the resulting controlled system.

The use of *forward* reachability can be a solution to both problems. Forward reachability starts with the initial states, and adds states reachable via one transitions, then via two transitions, then via three transitions, etc (although it may not be a breadth-first search, as it is configurable). This is repeated until all reachable states are found.

By combining both forward and backward reachability, the supervisor knows about states that exist in the uncontrolled system (due to forward reachability) and about states that it should forbid (due to backward reachability). This leads to the supervisor only restricting transitions that are strictly necessary. However, both when using forward reachability and when not using it, the synthesized supervisor is safe, non-blocking, and maximally permissive. It is only the guards that

are more complex than they might need to be, if forward reachability is not used. More complex guards are often less readable, and potentially more expensive to implement in an actual controller.

By combining both forward and backward reachability, parts of the state space that are not relevant may not have to be expanded (as much), which may improve performance. However, computing the forward reachability may also take time and cost memory, thus reducing performance. Note that the performance impact can differ based on when forward reachability is performed. By default, forward reachability is performed last, but this can be changed by configuring the order of the fixed-point computations.

It depends on the specification being synthesized whether enabling forward reachability increases or decreases performance. It also depends on the specification whether there is any benefit to using forward reachability for the guards of the supervisor. Forward reachability is disabled by default. It can be enabled using the *Forward reachability* option (see the options section). The order in which the edges of the specification are considered during reachability computations also influences the performance (see the edge order section), as does the edge granularity.

Fixed-point computations order

Data-based synthesis essentially works by computations that involve predicates that partition the entire state space into states that satisfy a property or don't satisfy a property. For instance, a marker predicate may indicate which states of the state space are marked. All other states are thus not marked.

The main computations performed during synthesis involve reachability computations. For instance, from which states is it possible to reach a marker state? To compute the states that can reach a marker state, the marker predicate of the input specification is used. The marker predicate indicates the states that are themselves marked. Then, the states are calculated that can reach one of those marked states, via a single transition. They are put together, to form the states that are marked or can be marked after one transition. By taking another such step, we can add the states that can reach a marked state via two transitions. We then have all states that can reach a marked state via zero, one, or two transitions. We can repeat this until no new states are found, which is called reaching a fixed point. The order in which transitions are considered and states are found may however not be exactly as described, as it is configurable.

The form of reachability described above is called *backward* reachability, as it starts from some target (e.g. marked states), and goes backwards to find all states from which the target can be reached. Conversely, *forward* reachability starts from the initial states, and adds states reachable via transitions until all reachable states are found.

Multiple fixed-point reachability computations are performed during synthesis:

- *Computing the non-blocking states*: The non-blocking states are those states that can reach a marked state. They are computed by a backward reachability computation, starting from the specification's marker predicate. This computation is mandatory, to ensure that a valid supervisor is computed.

- *Computing states that satisfy the reachability requirements*: A reachability requirement specifies a goal predicate that represents a set of states, and may restrict the source states, states allowed on the path, and number of transitions to the goal. Synthesis ensures that each applicable state in the controlled system can reach a goal state subject to these restrictions. Similar to the marker predicate, a backward reachability computation is used, starting from the reachability requirement's goal predicate and considering any restrictions. For each reachability requirement, a separate backward reachability computation is performed. If the specification contains reachability requirements, these computations are mandatory to ensure that a valid supervisor is computed. For bounded reachability requirements, edges are applied in explicit rounds to track the maximum number of transitions to the goal, using a dedicated exploration algorithm. The configured exploration strategy is therefore ignored for bounded reachability requirements.
- *Computing the controllable states*: The controllable states are those states where only controllable events are restricted. They are computed by inverting the result of a backward reachability computation over only the uncontrollable events, starting from the bad states. This computation is mandatory, to ensure that a valid supervisor is computed.
- *Computing the reachable states*: The reachable states are those states that can be reached from an initial state. They are computed by a forward reachability computation, starting from the initial states. Performing a forward reachability computation is optional, and can be enabled or disabled.

The enabled fixed-point computations are repeated until an overall fixed point is reached.

The order in which the different fixed-point computations are performed has no effect on the correctness of the resulting supervisor. However, it can have a significant impact on the performance of synthesis. For instance, if a model has a limited set of initial states and not many states can be reached via forward reachability, then computing the reachable states first may be beneficial. The limited set of reachable states may then impact a subsequent non-blocking computation, which only needs to consider those reachable states, which could reduce the effort needed to complete the computation. It may also reduce the size of the predicates produced during the fixed-point computation, which may reduce the amount of memory required to complete the computation and subsequent computations. However, if there are many initial states and the set of marked states is much smaller, the inverse may be true, and it could be better to start with the non-blocking computation. The best order, the order that gives the best performance, depends on the specification being synthesized.

By default, the non-blocking states computation is performed first, then the reachability requirement computations are performed, then the controllable states computation, and lastly the reachable states computation (if enabled). The order in which the fixed-point computations are performed can be configured using the *Fixed-point computations order* option. When specifying the value of the option on the command line, specify `nonblock` for the non-blocking states computation, `reach-reqs` for the reachability requirement computations, `ctrl` for the controllable states computation, and `reach` for the reachable states computation, in the desired order, joined by commas. For instance, specify `ctrl,reach-reqs,reach,nonblock` or `reach,reach-reqs,ctrl,nonblock`. Note that all four fixed-point computations must be included when specifying their order, regardless of whether they are all enabled or not.

For backward compatibility, the old syntax of the *Fixed-point computations order* is still supported.

This old syntax should be avoided, as support for it will be removed in a future release. The following table indicates for each value in the old syntax, its corresponding value in the new syntax:

Old syntax	New syntax
`nonblock-ctrl-reach`	`nonblock,reach-reqs,ctrl,reach`
`nonblock-reach-ctrl`	`nonblock,reach-reqs,reach,ctrl`
`ctrl-nonblock-reach`	`ctrl,nonblock,reach-reqs,reach`
`ctrl-reach-nonblock`	`ctrl,reach,nonblock,reach-reqs`
`reach-nonblock-ctrl`	`reach,nonblock,reach-reqs,ctrl`
`reach-ctrl-nonblock`	`reach,ctrl,nonblock,reach-reqs`

Besides the order of the fixed-point computations, enabling or disabling the forward reachability computation can also have a significant impact on the synthesis performance.

Input variables

Data-based synthesis supports input variables. The model itself doesn't specify which value an input variable has at any given moment. Input variables can thus have any value (as long as it fits within the data type of the variable), and the value can change at any time. Input variables are ideal to model sensors.

To support this for data-based synthesis, the input variable is treated as a discrete variable with an arbitrary initial value. To allow the input variable to arbitrarily change, an uncontrollable event is added (with the same absolute name as the input variable). Also, a single edge is added for that event. The edge is always enabled (guard `true`, since the input variable can always change value), and the update indicates that it can get any value that it doesn't currently have (`x+ != x` for `x` an input variable, with `x` the value of the variable before the update, and `x+` the value of the variable after the update). Obviously, the value of the input variable is kept within the range of values that is allowed by its data type.

Using synthesis with requirements that restrict the allowed values of an input variable will result in an empty supervisor, as a supervisor can't prevent the environment from changing the value of the input variable (it would have to restrict the uncontrollable event that is internally added to model value changes of the input variable). A supervisor can however impose additional restrictions on the initial value of an input variable. The supervisor can then only guarantee safe, non-blocking behavior if the system is initialized in accordance with the additional initialization restrictions.

Initialization

The data-based synthesis algorithm ensures that in the controlled system, the state requirement invariants hold in all reachable states. It also ensures that in the controlled system, for all transitions from reachable states, the events only occur if the requirement automata and

state/event exclusion invariants allow them.

The synthesis algorithm does not restrict any uncontrollable events. Instead, such restrictions are propagated backwards to the source state of the edge with the uncontrollable event, and from there to the transitions that lead to the source state, etc. They are propagated backwards until an edge with a controllable event is encountered (for which the guard can be restricted) or the initial state is reached (and the initialization predicate can be restricted).

If a variable in the uncontrolled system has a single initial value, and the initialization predicate is restricted to not allow this initial value, initialization will be impossible, causing an *empty supervisor* error. For discrete variables with multiple potential initial values, the synthesis algorithm may restrict initialization to disallow certain initial values, while still leaving possibilities for initialization. For discrete variables declared to initially have an arbitrary initial value, as well as for input variables, the synthesis algorithm essentially determines under which conditions the system can be started, and still exhibits only safe, non-blocking behavior.

If the controlled system requires more strict initialization than the uncontrolled system, an additional initialization predicate is added to the resulting supervisor. The exact predicate may differ, depending on the simplifications that are applied.

Performance

There are quite a few factors that influence the performance of data-based synthesis. Here are some of those factors, which affect both the time it takes to complete a synthesis and the memory that is used for it:

- Obviously, the actual model that is used has a large impact. A model with more variables often has to longer synthesis time and requires more memory. However, the predicates that are used may also significantly impact performance.
- The modeling style used when modeling the plants and requirements can also have a significant impact on the performance. See for instance these suggestions.
- Several options of the data-based synthesis tool impact its performance. A list of such option is given below.

If you run out of memory, here are some potential solutions:

- Increase the memory available to the synthesis tool. For more information, see the page of the Eclipse ESCET general toolkit documentation on resolving performance and memory problems.
- Decrease the size of the BDD operation caches, to make sure more memory remains available for the actual data used during synthesis. While decreasing the size of caches may increase the synthesis time, it is better to be able to synthesize a supervisor in a longer time than to not be able to synthesize it at all.

If synthesis is slow, the following options are easy to use and often have significant impact on the synthesis time:

- If you don't care about the readability of the resulting supervisor, consider setting the BDD

output mode to `nodes`.

- Depending on the model, it can help significantly if forward reachability is enabled, and is performed as the first computation in the fixed-point computation order.
- Be careful with enabling the 'Maximum used BDD nodes' statistic, as enabling it generally makes synthesis much slower.

Performance-related options

The following options have an effect on the performance of data-based synthesis:

Kind	Option	Section	Effect	Choose
Input	State requirement invariant enforcement	State requirement invariants	Impacts the up-front effort, size of the controlled behavior, number of iterations to reach fixed points, and the post-synthesis effort	Complex trade-off between different effects, depends on the model
Input	Edge granularity	Edge granularity	Impacts the size of the edge representations, size of the controlled behavior, number of iterations to reach fixed points, and the post-synthesis effort	Choose the best representation, depends on the model
Output	BDD output mode	BDD representation in CIF	Representation of BDDs in the output model	Use `nodes` output variant for best performance
Output	BDD predicate simplify	Predicate simplification	Potentially smaller BDDs in the output	Enable for smaller output, although simplification itself also takes time
Order	Fixed-point computations order	Fixed-point computations order	Better order for less computations and smaller BDD representations	Choose the best ordering, depends on the model
Order	Exploration strategy	Edge order	Better order for less computations and smaller BDD representations	The `saturation` strategy typically gives the best performance

Kind	Option	Section	Effect	Choose
Order	BDD initial variable ordering	Initial variable orders	Better order for smaller BDD representations	Choose the best ordering, depends on the model, (reversed) model/sorted usually good choices, custom order allows for best performance
Order	BDD hyper-edge creation algorithm	Variable relations	Better order for smaller BDD representations	Choose the algorithm that produces the best order, depends on the model
Order	BDD hyper-edge graph creation algorithm	Automatic variable ordering	Faster computation for dense variable relations	The `dense` algorithm likely works best for dense and large models
Order	BDD DCSH variable ordering algorithm	Variable orderers	Better order for smaller BDD representations	Enable for automatic ordering
Order	BDD FORCE variable ordering algorithm	Variable orderers	Better order for smaller BDD representations	Enable for automatic ordering
Order	BDD sliding window variable ordering algorithm	Variable orderers	Better order for smaller BDD representations	Enable for automatic ordering
Order	BDD sliding window size	Variable orderers	Better order for smaller BDD representations	Larger windows might allow for more optimization, but take more time
Order	BDD advanced variable ordering	Advanced variable ordering configuration	Better order for smaller BDD representations	Choose the best ordering, depends on the model
Order	Edge order for backward reachability	Edge order	Better order for less computations and smaller BDD representations	Choose the best order, depends on the model, custom order allows for best performance
Order	Edge order for forward reachability	Edge order	Better order for less computations and smaller BDD representations	Choose the best order, depends on the model, custom order allows for best performance

Kind	Option	Section	Effect	Choose
Library	BDD library initial node table size	Options	More storage for less resizes	Increase size for less resizes, at the cost of less memory locality
Library	BDD library operation cache size/ratio	BDD operation cache	Increase cache for less computations	Enable, larger costs more memory, larger leads to less memory locality, size/ratio depends on model
Algorithm	Forward reachability	Forward reachability	Explore only reachable state space	Enable to reduce state space, although calculation itself may also be expensive, depends on model
Debug	Output mode	Debug output	Debug output on console	Disable for better performance
Debug	BDD debug max nodes/paths	Debug output	Size of predicates in debug output	The smaller, the less blowup, the better the performance
Debug	Statistics	Statistics	Statistics output on console or to file	Disable for better performance
Warnings	Event warning	Early problem detection and Resulting supervisor	Warning for never enabled events	Disable for better performance
Warnings	Plants referencing requirements warnings	Early problem detection	Warning for plants that reference requirement state	Disable for better performance

The first column categorizes the different options a bit, for different kind of options. The second column lists the different options. The third column indicates in which section of the data-based synthesis tool documentation you can find more information about that option. The fourth column indicates the effect of the option. The fifth column indicates what to choose for the option, for best performance, although a trade-off may be involved.

Edge granularity

To prepare for synthesis, the data-based synthesis tool first linearizes the specification. This essentially combines the edges of the various automata in different ways, to form new self-loop edges that are then part of a single location of a single automaton. After that, each such linearized edge is converted to a Binary Decision Diagram (BDD) representation, a symbolic representation of the edge that allows efficient computations. The BDD representations of the edges are then used to perform the actual synthesis.

However, using the linearized edges 'as is' doesn't always give the best synthesis performance. It may be beneficial to combine some linearized edges together, to form a single combined edge. This combined edge may have a smaller BDD representation than the original edges from which it was combined, reducing memory usage during synthesis. The combined edge may also lead to faster convergence for reachability computations, reducing both memory usage and synthesis time. However, the effect depends on the particular model being synthesized, and combining edges can also lead to an increase in memory usage and synthesis time. Furthermore, the synthesis performance may also be affected by the order in which the edges are considered during reachability computations, and whether or not forward reachability is used.

The edge representation to use for synthesis can thus be more granular (more edges) or less granular (less edges). The following granularities are supported:

- *Per edge*: Allow each event to have multiple edges. Essentially, the linearized edges are used 'as is'.
- *Per event*: Ensure each event has exactly one edge. Essentially, for each event, the linearized edges are merged together to form a single edge for that event.

The granularity can be configured using the *Edge granularity* option (see the options section). By default, the 'per event' edge granularity is used.

Linearization and non-determinism

The CIF data-based synthesis tool uses the 'product with merge' variant of linearization. This variant ensures that all behavior of the specification is preserved, while it doesn't blow up as much in the number of linearized edges as the 'product' variant does. After having merged the edges for a certain event together into a single event, one may wonder whether the result is then the same as would have been obtained by the 'merge' variant of linearization. This is indeed the case in many situations, but there are exceptions.

If two linearized edges (as obtained by the 'product with merge' variant) have overlapping guards, they can be enabled at the same time, and we therefore have to deal with this non-determinism. If the edges then have different updates, we get to the exceptional case. In such cases, the 'merge' variant will choose one of the updates, while the approach for merging edges on BDD representation as is used for 'per event' edge granularity will allow both updates. Note that there is no way in CIF to represent such merged edges as a single CIF edge, but such edges can be represented using BDDs.

Example

As an example, consider two linearized edges, obtained by the 'product with merge' variant:

- `when x <= 4 do x := x + 1`
- `when x >= 4 do x := x - 1`

When we combine them, we first combine their guards using `or`. This gives us `x <= 4 or x >= 4`, which can be simplified to `true`. This is a good example of how combining edges may simplify them, reducing their memory usage.

We then also combine their updates, where we distinguish 3 cases:

- Case 1 (the first guard holds, the second one doesn't): Here `x <= 4 and not x >= 4`, which is `x <= 4 and x < 4`, which is just `x < 4`. As only the first edge is enabled, we get that if guard `x < 4` holds, then update `x := x + 1` should be applied.
- Case 2 (the second guard holds, the first one doesn't): Here `not x <= 4 and x >= 4`, which is `x > 4 and x >= 4`, which is just `x > 4`. As only the second edge is enabled, we get that if guard `x > 4` holds, then update `x := x - 1` should be applied.
- Case 3 (both guards hold): Here `x <= 4 and x >= 4`, which is just `x = 4`. As both edges are enabled, we get that if guard `x = 4` holds, then either of the updates may be applied.

Case 3 can not be directly represented as an update of a single edge of a CIF model, as on a single edge, even using `if` updates, we can not perform two different updates on the same variable. This is a good example of how the updates can become more complex, as we need to distinguish various cases, and relate the updates to the guards and combinations of guards, which may be more complex then the original edges that were combined.

Edge order

Data-based supervisor synthesis involves many reachability computations to determine which states can be reached from which other states (forward reachability), or which states can reach which other states (backward reachability). This involves repeatedly following edges to find those states, using a fixed point computation.

Synthesis does not directly use the edges from the model. It applies linearization and then uses the edges from the linearized model. Edges are internally also added to support input variables during synthesis. Furthermore, edges may be combined to form less granular edges, before they are used for reachability computations.

During reachability computations the edges are then considered one by one to find more reachable states. As long as new states are found, the edges will keep being followed. Once it is clear that considering edges no longer leads to finding new states, a fixed point has been reached and the computation stops.

The order in which the edges are considered for such computations has no effect on the final fixed point result. The order can however significantly influence the performance of supervisor synthesis. For instance, consider that there are dozens of edges and from the already discovered states there is only one edge that leads to more states being found. Then considering that one particular edge first will immediately lead to more states being found. Considering other edges first will lead to wasted computations, as they won't find any new states.

Besides finding or not finding new states, which new states are found first and in what order may also influence performance. It may lead to smaller or larger intermediate BDD representations of predicates. See the section about BDD variable order for more information.

Strategies

There are several strategies that determine the order in which edges are considered. Which strategy to use during synthesis, can be configured using the *Exploration strategy* option (see the options section). The strategy has no effect on bounded backward reachability for reachability requirements, which uses a dedicated round-based algorithm.

The following strategies are available. After each strategy, the value to specify on the command line to select that strategy is listed between parentheses:

- *Fixed order* (`chaining-fixed`):

  A fixed order is determined upfront, before any reachability computations. Each of the edges is considered in that fixed order, one after the other. Regardless of whether an edge leads to new states being found or not, the search always continues with the next edge from the fixed order. Each edge is applied to all reached states, including those of the previously applied edges. After the last edge has been considered, the first edge is considered again, then the second one, etc. This way of applying edges is known as the chaining strategy for reachability computation. A fixed point is reached once:

  - All edges have been considered at least once.
  - Since all edges have last been considered, no new states were found.

- *Workset algorithm* (`chaining-workset`):

  The order in which the edges are considered is dynamic, rather than fixed. The algorithm also follows the chaining exploration strategy, and works as follows:

  - Initially, all edges are in a workset.
  - While the workset is not empty:
    - Select an edge from the workset. In principle any edge may be selected, but using a smart heuristic improves performance.
    - Use the selected edge to find new states, repeatedly, until it no longer has an effect (no longer finds new states).
    - If any new states were found using the selected edge (it had an effect), add the dependencies of that edge to the workset. The dependencies of an edge are those other edges that may be enabled after using it.
    - Remove the selected edge from the workset.

  The workset algorithm can only be used in combination with a 'per event' edge granularity. The workset algorithm can not be used when duplicate events are allowed in a custom edge order.

- *Saturation* (`saturation`):

  Saturation is a strategy for BDD-based symbolic reachability computations that exploits locality of the edges that are considered. The edges to consider are sorted on the top-most BDD variable of their BDD representations. The order of edges with the same such top-most BDD variable is determined based on the configured order. The saturation algorithm then uses this order to saturate the BDD representation of the set of reachable states from the bottom up. This is done

by first applying edges over the lowest part of the BDD until they no longer lead to new states, and then gradually go upward in the sorted edge order until saturation is reached for all edges. Saturation thereby tries to reduce blow-ups of BDDs at intermediate points during reachability computations, which can sometimes be observed with other exploration strategies, like chaining.

The saturation strategy can not be used when duplicate events are allowed in a custom edge order.

Saturation is used by default.

Which strategy works best depends on the model, and this may involve a trade-off. The workset algorithm may reduce the maximum amount of memory used during synthesis, compared to using a fixed order. However, this often comes at the cost of extra computations that require more time, especially if edges have many other edges as their dependencies. Saturation generally works best.

Configuring the order

The order in which the edges are considered when a fixed order is used, is determined by the *Edge order for backward reachability* and *Edge order for forward reachability* options (see the options section). If the workset algorithm is used, the edges in the workset are ordered according to the configured order. The configured order may therefore also influence the order in which edges are selected by the workset algorithm's edge selection heuristics. And if saturation is used, the configured edge order determines the order of the edges with the same top-most BDD variable.

Several predefined orders exist, and it is also possible to define a custom order. By default, the *model ordering* is used for both backward and forward reachability computations. The following orders can be used:

- *model ordering* (option value `model`)

  The order of the edges is as they occur in the linearized model. Edges for input variables are always put after the other edges, sorted based on the variable order. Edges are considered exactly once per reachability iteration.

- *reverse model ordering* (option value `reverse-model`)

  The order of the edges is as they occur in the linearized model, but reversed. Edges for input variable are thus always put before the other edges, sorted based on the reversed variable order. Edges are considered exactly once per reachability iteration.

- *sorted ordering* (option value `sorted`)

  The order of the edges is based on the names of their corresponding events and input variables. They are sorted alphabetically in ascending order, based on their absolute names. In case edges are labeled with the same event, the edges are ordered based on the linearized model order. Edges are considered exactly once per reachability iteration.

- *reverse sorted ordering* (option value `reverse-sorted`)

  The order of the edges is based on the names of their corresponding events and input variables.

They are sorted alphabetically in descending order, based on their absolute names. In case edges are labeled with the same event, the edges are ordered based on the reversed linearized model order. Edges are considered exactly once per reachability iteration.

- *random ordering* (option value `random` or `random:SEED`)

  The edges are ordered randomly. If no seed is specified, a random seed is used, resulting in a random order. If a seed is specified, a fixed random order is used. That is, using the same seed again, results in the same random order. The `SEED` must be an integer number in the range [0 .. 2^{64} - 1]. For instance, use `random:123` as option value to get a random order that can be repeated on a subsequent synthesis for the same model.

- *custom ordering*

  Custom orders consist of absolute names of events and input variables. That is, for an automaton `a`, with an event `x`, the absolute name of the event is `a.x`. In case edges are labeled with the same event, the edges are ordered based on the linearized model order. The `*` character can be used as wildcard in those names, and indicates zero or more characters. In case of multiple matches, the matches are sorted alphabetically in ascending order, based on their absolute names.

  Multiple names can be separated with `,` characters. The edges and input variables matching the name pattern before the `,` are ordered before the edges and input variables matching the name pattern after the `,`.

  Each name pattern in the order must match at least one event or input variable. Every event and input variable needs to be included in the order.

  By default, an event or input variable may not be included more than once in the order. This allows detecting accidental duplicate inclusion of events in the order. It can however be useful to include events multiple times in the order, for instance if for some model considering the event twice leads to immediately finding a new state during the reachability operation, while this would otherwise only be found in the next iteration. To allow events to be included multiple times in the custom order, use the *Edge order duplicate events* option (see the options section). This option applies to both the backward and forward edge order.

Determining the best edge order is difficult as it can be tricky to predict which edges will lead to finding new states and quickly reaching the fixed point result. When in doubt, keep the default values of the options. If you specify a custom edge order for backward reachability, typically the reverse order is a decent order for forward reachability, and vice versa.

BDD variable order

Internally, data-based synthesis represents predicates using Binary Decision Diagrams (BDDs). CIF variables and automata are represented using one or more boolean variables (also called BDD variables or bits). For instance, a boolean CIF variable is represented using a single boolean/BDD variable, and a CIF variable of type `int[0..8]` is represented using four boolean/BDD variables (9 possible values, $\log_2(9) \approx 3.17$). For each automaton with two or more locations, a location pointer

variable is created, that represents the current or active location of that automaton. For instance, an automaton with three locations is represented using two boolean/BDD variables. Two boolean/BDD variables can represent $2^2 = 4$ values, so one value is not used.

The CIF variables and location pointer variables for the automata (together called synthesis variables) can be ordered. If a CIF variable has a tuple type, it fields are (recursively) represented as separate synthesis variables. The ordering of synthesis variables can significantly influence the performance of synthesis. Synthesis variables that have a higher influence on the result of predicates (simply put, occur more frequently in predicates) should generally be put earlier in the ordering. Furthermore, in general, strongly related synthesis variables (e.g. by comparison, integer computation, or assignment) should be kept closely together in the order. For two synthesis variables `x` and `y`, examples of predicates that introduce relations are `x = y` (by comparison) and `5 < x + y` (by integer computation), and examples of assignments that introduce relations are `x := y` and `x := y + 1` (both by assignment).

In certain cases, it is not only possible to order the BDD variables per synthesis variable, but also to interleave the BDD/boolean variables of some synthesis variables. This can significantly influence the performance of synthesis. Generally, strongly related synthesis variables should be interleaved.

For each (part of a) CIF variable and location pointer, two synthesis variables are created, one storing the old/current value (before a transition), and one storing the new value (after a transition). For a single (part of a) CIF variable or location pointer, the old and new synthesis variables are always kept together, and interleaved. The old synthesis variable is also always before the new synthesis variable.

For more information on variable ordering and its influence on performance, see Chapter 3 of [Minato (1996)].

The following additional information is available on variable ordering configuration through options:

- Variable ordering options
- Basic
  - Initial variable orders
  - Automatic variable ordering
- Advanced
  - Advanced variable ordering configuration

Variable ordering options

The variable ordering used by the CIF data-based synthesis tool can be configured through options. There are two ways to configure variable ordering:

- Basic configuration using basic options.
- Advanced configuration using an advanced option.

These options allow configuring the variable order to one of several predefined orders, or to a custom variable order. Manually specifying a custom order often requires specialist knowledge and

can take quite some time. Luckily, there are algorithms that can automatically compute a decent variable order by improving an existing variable order. The algorithms to apply, and their settings, can be set using options as well.

The following basic options are available for basic configuration of variable ordering:

- *BDD initial variable ordering*
- *BDD hyper-edge creation algorithm*
- *BDD hyper-edge graph creation algorithm*
- *BDD DCSH variable ordering algorithm*
- *BDD FORCE variable ordering algorithm*
- *BDD sliding window variable ordering algorithm*
- *BDD sliding window size*

For more information on the use of these basic options, see:

- Initial variable orders
- Automatic variable ordering

For more flexibility, variable ordering can instead be configured using an advanced option:

- Advanced variable ordering configuration

Variable ordering may be configured using either the basic options or using the advanced option. It is not supported to configure variable ordering using both basic and advanced configuration at the same time.

Initial variable orders

The CIF data-based synthesis algorithm features several predefined initial BDD variable orders, and it is also possible to define a custom order.

For basic configuration, the initial order of the boolean/BDD variables can be configured using the *BDD initial variable ordering* option (see the options section). By default, the sorted order is used as initial variable ordering. The following predefined initial orderings can be used:

- *model ordering without interleaving* (option value `model`)

  The order of the synthesis variables is as they occur in the model. A location pointer, for an automaton with two or more locations, is put before the variables declared in that automaton.

  No synthesis variables are interleaved, except for each old variable with its corresponding new variable.

- *reverse model ordering without interleaving* (option value `reverse-model`)

  The order of the synthesis variables is as they occur in the model, but reversed. A location pointer, for an automaton with two or more locations, is put after the variables declared in that

automaton, in this reverse order.

No synthesis variables are interleaved, except for each old variable with its corresponding new variable. The old variables are still before the new variables; this is not reversed.

- *sorted ordering without interleaving* (option value `sorted`)

  The order of the synthesis variables is based on the names of the variables and automata. They are sorted alphabetically in ascending order, based on their non-escaped absolute names.

  For tuple-typed CIF variables, a synthesis variable is created for each field of the tuple. The ordering of the fields is based on their non-escaped names.

  No synthesis variables are interleaved, except for each old variable with its corresponding new variable.

- *reverse sorted ordering without interleaving* (option value `reverse-sorted`)

  The order of the synthesis variables is based on the names of the variables and automata. They are sorted alphabetically in descending order, based on their non-escaped absolute names.

  For tuple-typed CIF variables, a synthesis variable is created for each field of the tuple. The ordering of the fields is based on their non-escaped names.

  No synthesis variables are interleaved, except for each old variable with its corresponding new variable. The old variables are still before the new variables; this is not reversed.

- *random ordering without interleaving* (option value `random` or `random:SEED`)

  The synthesis variables are ordered randomly. If no seed is specified, a random seed is used, resulting in a random order. If a seed is specified, a fixed random order is used. That is, using the same seed again, results in the same random order. The `SEED` must be an integer number in the range [0 .. 2^{64} - 1]. For instance, use `random:123` as option value to get a random order that can be repeated on a subsequent synthesis for the same model.

  No synthesis variables are interleaved, except for each old variable with its corresponding new variable.

- *custom ordering*

  Custom orders consist of absolute non-escaped names of synthesis variables. For instance, for an automaton `a`, with a discrete variable `x`, the absolute name of the variable is `a.x`. If a synthesis variable represents a part of a tuple-typed CIF variable, projections need to be specified to access that part, using the field names. For instance, for an input variable named `x` of type `tuple(tuple(bool a, b) c, d)`, you would specify `x[c][a]`, `x[c][b]`, `x[d][a]` and `x[d][b]`. The `*` character can be used as wildcard in those names, and indicates zero or more characters. In case of multiple matches, the matches are sorted increasingly on their non-escaped absolute names, and interleaved.

  Multiple names can be separated with `;` characters. The synthesis variables matching the name pattern before the `;` are ordered before the synthesis variables matching the name pattern after

the `;`. The `;` separator does not introduce interleaving. The `,` separator can be used instead of the `;` separator to introduce order but also introduce interleaving.

Each name pattern in the order must match at least one synthesis variable. A synthesis variable may not be included more than once in the order. Every synthesis variable needs to be included in the order. It is not possible to specify new variables, as they are always directly after their corresponding old variables, and they are always interleaved.

For instance, consider two automata: `a` and `b`, each with three variables of type `int[0..3]`: `x1`, `x2`, and `x3`. The automata have three locations each, which means location pointers are created for them. We thus have six discrete variables: `a.x1`, `a.x2`, `a.x3`, `b.x1`, `b.x2`, and `b.x3`, and two location pointer variables: `a` and `b`. Consider the following custom order: `b*;a.x3,a.x1;a.x2,a`. Pattern `b*` matches location pointer variable `b` as well as the three discrete variables of automaton `b` (`b.x1`, `b.x2`, and `b.x3`). They are ordered in increasing alphabetic order, and are interleaved. Variables `a.x3` and `a.x1` are also interleaved, with `a.x3` before `a.x1`. Finally, variable `a.x2` is ordered before the location pointer for automaton `a`, and they are interleaved as well. This results in the following boolean/BDD variable ordering, with bits whose name ends with `+` representing bits of new variables rather than current/old variables, and `x#0` representing bit zero of variable `x`:

```
b#0
b+#0
b.x1#0
b.x1+#0
b.x2#0
b.x2+#0
b.x3#0
b.x3+#0
b#1
b+#1
b.x1#1
b.x1+#1
b.x2#1
b.x2+#1
b.x3#1
b.x3+#1

a.x3#0
a.x3+#0
a.x1#0
a.x1+#0
a.x3#1
a.x3+#1
a.x1#1
a.x1+#1

a.x2#0
a.x2+#0
a#0
a+#0
```

```
a.x2#1
a.x2+#1
a#1
a+#1
```

For advanced configuration, see the separate section on advanced variable ordering configuration.

Automatic variable ordering

The CIF data-based synthesis tool's predefined initial variable orders are often not optimal performance-wise. Manually specifying a custom order often requires specialist knowledge and can take quite some time. Luckily, there are algorithms that can automatically compute a decent variable order.

The algorithms all take an existing variable order as input, and try to improve it using a fast heuristic. Some algorithms search for a local optimum. A better starting variable order may then result in a better final variable order (a better local optimum), and may also speed up the automatic variable ordering algorithm itself (reaching an optimum faster). Other algorithms search for a global optimum. However, the algorithms are all based on heuristics. The guarantees that they provide differ, but none of them guarantees that synthesis will actually be quicker. In practice however, using such algorithms can significantly improve synthesis performance, especially for larger and more complex models.

For the initial variable ordering, the CIF variables and location pointers may be arbitrarily interleaved. If an automatic variable ordering algorithm changes the initial order, no synthesis variables are interleaved, except for each old variable with its corresponding new variable.

The automatic variable ordering algorithms are not applied if the CIF model has less than two synthesis variables, as with zero variables there is nothing to order, and with one variable there is only one possible order. They are also not applied if the model has no guards, updates, or other predicates from which to derive relations between the variables. Without such relations, the algorithms lack the required input to search for improved variable orders.

Variable relations

The variable relations serve as input for the algorithms. Different algorithms may use different representations of the variable relations. One such representation is *hyper-edges*, and another is *graph edges*, which are derived from the hyper-edges.

For basic configuration, the variable relations that are used to construct the hyper-edges can be configured using the *BDD hyper-edge creation algorithm* option (see the options section). By default, the *Default* relations are used. The following variable relations can be used:

- *Legacy*

  The legacy hyper-edge creator creates the following hyper-edges:

  - Per invariant, a hyper-edge for the synthesis variables that occur in the invariant.
  - Per edge in an automaton, per guard, per comparison binary expression, a hyper-edge for the synthesis variables that occur in the binary expression.

    - Per assignment, a hyper-edge for the synthesis variables that occur in the addressable and value of the assignment.
    - Per event, a hyper-edge for the synthesis variables that occur in the guards and updates of all edges for that event in the entire specification.

- *Linearized*

  The linearized hyper-edge creator creates the following hyper-edges:

    - Per linearized edge, a hyper-edge for the synthesis variables that occur in the guards and updates of that linearized edge. State/event exclusion invariants are taken into account, by first converting them to automata.

- *Default*

  Uses the linearized hyper-edge creator for the *FORCE* and *sliding window* variable ordering algorithms, and the legacy hyper-edge creator otherwise.

All hyper-edge creators take into account synthesis variables that occur indirectly via algebraic variables.

The graph edges can be constructed in three distinct ways using the *BDD hyper-edge graph creation algorithm* option (see the options section). The graph creation algorithm primarily affects the time it takes to construct the graph itself, but it can also influence the variable order, as it impacts the order in which variable ordering algorithms consider the graph edges. By default, the *Sparse* graph creation algorithm is used. The following graph creation algorithms can be configured:

- *Sparse*

  The sparse graph edge creation algorithm creates the graph using a map-based method, which is optimised for sparse variable relations.

- *Dense*

  The dense graph edge creation algorithm creates the graph using a matrix-based method, which is optimised for dense variable relations. This algorithm is not supported for CIF specifications with more than 65,535 synthesis variables.

- *Auto*

  The automatic graph edge creation algorithm creates the graph using either the sparse or dense algorithm, based on the average hyper-edge cardinality.

More detailed information about the various representations of variable relations may be obtained during synthesis by enabling debug output.

For advanced configuration, see the separate section on advanced variable ordering configuration.

Variable orderers

The following variable ordering algorithms, or variable orderers, are available:

- *DCSH*

  The DSM-based Cuthill-McKee-Sloan variable ordering Heuristic, or DCSH algorithm, of [Lousberg et al. (2020)] aims to find good global variable orders.

  DCSH applies two algorithms, the Weighted Cuthill-McKee bandwidth-reducing algorithm and the Sloan profile/wavefront-reducing algorithm. It then chooses the best order out of the orders produced by these two algorithms as well as their reverse orders, based on the Weighted Event Span (WES) metric.

  The DCSH algorithm is enabled by default. For basic configuration, it can be disabled using the *BDD DCSH variable ordering algorithm* option (see the options section).

- *FORCE*

  The FORCE algorithm of [Aloul et al. (2003)] aims to optimize an existing variable order, but may get stuck in a local optimum.

  FORCE iteratively computes new variable orders by considering relations between the variables. Conceptually, highly related variables 'pull' each other closer with more force than less related variables do. Each new order is promoted as the new best order, if it is better than the current best order, based on the total span metric. The iterative algorithm stops once a fixed point has been reached, or after at most 10 * ceil($\log_e$(n)) iterations of the algorithm have been performed, with `n` the number of synthesis variables.

  The FORCE algorithm is enabled by default. For basic configuration, it can be disabled using the *BDD FORCE variable ordering algorithm* option (see the options section).

- *Sliding window*

  The sliding window algorithm aims to locally optimize an existing variable order for each window.

  The sliding window algorithm 'slides' over the variable order, from left to right, one variable at a time, using a fixed-length *window*. A window is a part of the entire order. For instance, for a variable order `a;b;c;d;e` and windows length 3, the window at index 0 would be `a;b;c`, at index 1 it would be `b;c;d`, and at index 2 it would be `c;d;e`. For each window, it computes all possible variable permutations, and selects the best one based on the total span metric. It then replaces the window by the best permutation of that window, before moving on to the next window.

  The sliding window algorithm is enabled by default. For basic configuration, it can be disabled using the *BDD sliding window variable ordering algorithm* option (see the options section).

  The default maximum length of the window that is used is 4. The actual window may be smaller, if less than 4 variables and/or location pointers are present in the model. For basic configuration, the maximum length of the window can be configured using the *BDD sliding window size* option (see the options section). The option to set the maximum length only has effect if the sliding window variable ordering algorithm is enabled. The size must be an integer number in the range [1 .. 12].

If enabled using basic configuration, the algorithms are applied in the order they are listed above.

For advanced configuration, see the separate section on advanced variable ordering configuration.

Advanced variable ordering configuration

The CIF data-based synthesis performance depends greatly on the variable ordering. After selecting an initial ordering, it can subsequently be improved by algorithms for automatic variable ordering. This may be configured using basic options, but they offer only limited flexibility. For the most flexibility, variable ordering can be configured using the *BDD advanced variable ordering* option (see the options section). Advanced ordering allows configuring which orderers, such as predefined orders and ordering algorithms, to use. It also allows configuring the order in which the various orderers are applied, as well as the settings to use for each orderer.

Variable ordering may be configured using either the basic options or using the advanced option. It is not supported to configure variable ordering using both basic and advanced configuration at the same time.

Examples

The predefined initial variable orderings can be expressed using the advanced syntax as follows:

Predefined initial ordering	Advanced syntax
`model`	`model`
`reverse-model`	`model(effect=var-order) -> reverse(effect=both)`
`sorted`	`sorted`
`reverse-sorted`	`sorted(effect=var-order) -> reverse(effect=both)`
`random`	`random`
`random:SEED`	`random(seed=SEED)`
`<custom-order>`	`custom(order="<custom-order>")`

Some other examples of advanced variable orderings:

- `model -> dcsh -> force -> slidwin`: Start with model order as initial order, and then apply the DCSH, FORCE and sliding window algorithms with their default settings.
- `sorted -> dcsh(relations=linearized)`: Start with sorted order as initial order, and then apply the DCSH algorithm with linearized variable relations.
- `random -> or(choices=[dcsh,force])`: Start with a random order as initial order, and then apply the DCSH and FORCE algorithms, choosing the best order from the ones produced by the two algorithms.
- `force -> reverse`: Start with the default initial model order, then apply the FORCE algorithm, and finally reverse the variable order.

Syntax

The syntax to specify advanced variable ordering generally works as follows:

- Orderers are specified by their name.

- Arguments to configure orderers may optionally be given after the name of the orderer between parentheses: `(...)`.
- It is allowed to provide empty parentheses, e.g., `model()` is equivalent to specifying `model`.
- Each argument is given by a name and a value: `name=value`.
- To specify multiple arguments, separate them with a comma. After the last argument, optionally a comma may be specified as well.
- To apply multiple algorithms in sequence, separate them with an arrow (`->`).
- If needed, a sequence of orderers may be enclosed in parentheses: `(... -> ... -> ...)`.
- Spaces, tabs, and newline characters are generally ignored, but can be used to improve readability.

Orderers

The following orderers can be specified:

- *Basic ordering*

  Uses the configuration as configured by the basic variable ordering options.

  - Name: `basic`
  - Arguments: n/a
  - Example: `basic`
- *Model order*

  Uses the predefined model order.

  - Name: `model`
  - Arguments:
    - `effect`
      - Description: the effect of the variable orderer
      - Mandatory: no
      - Default: `both`
      - Type: enum (`var-order`, `representations` or `both`)
      - Constraints: none
  - Examples: `model`, `model(effect=var-order)`
- *Sorted order*

  Uses the predefined sorted order.

  - Name: `sorted`
  - Arguments:
    - `effect`

        - Description: the effect of the variable orderer
        - Mandatory: no
        - Default: `both`
        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
    - Examples: `sorted`, `sorted(effect=var-order)`
- *Random order*

  Uses the predefined random order.

    - Name: `random`
    - Arguments:
        - `seed`
            - Description: a seed for the random order
            - Mandatory: no
            - Default: use a random seed
            - Type: number
            - Constraints: seed must be in the range [0 .. 2^{64} - 1]
        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `both`
            - Type: enum (`var-order`, `representations` or `both`)
            - Constraints: none
    - Examples: `random`, `random(seed=123)`, `random(seed=123, effect=var-order)`
- *Custom order*

  Uses a custom order.

    - Name: `custom`
    - Arguments:
        - `order`
            - Description: the custom order, with the same syntax as for the basic configuration option
            - Mandatory: yes
            - Default: n/a
            - Type: string
            - Constraints: the order must be complete in that it contains all variables, and must not

contain any duplicate variables

        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `both`
            - Type: enum (`var-order`, `representations` or `both`)
            - Constraints: none
    - Examples: `custom(order="a,b;c,d")`, `custom(order="a,b;c,d", effect=var-order)`
- *DCSH*

  Applies the DCSH algorithm.

    - Name: `dcsh`
    - Arguments:
        - `node-finder`
            - Description: the node finder algorithm to use for the Weighted-Cuthill-McKee orderer
            - Mandatory: no
            - Default: `george-liu`
            - Type: enum (`george-liu` or `sloan`)
            - Constraints: none
        - `metric`
            - Description: the metric to use to compare orders
            - Mandatory: no
            - Default: `wes`
            - Type: enum (`total-span` or `wes`)
            - Constraints: none
        - `relations`
            - Description: the kind of relations to use when computing metric values
            - Mandatory: no
            - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
            - Type: enum (`configured`, `legacy` or `linearized`)
            - Constraints: none
        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `var-order`

        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
    - Examples: `dcsh`, `dcsh(metric=wes)`, `dcsh(node-finder=george-liu, metric=wes, relations=configured, effect=both)`
- *FORCE*

  Applies the FORCE algorithm.

    - Name: `force`
    - Arguments:
        - `metric`
            - Description: the metric to use to compare orders
            - Mandatory: no
            - Default: `total-span`
            - Type: enum (`total-span` or `wes`)
            - Constraints: none
        - `relations`
            - Description: the kind of relations to use when computing metric values
            - Mandatory: no
            - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
            - Type: enum (`configured`, `legacy` or `linearized`)
            - Constraints: none
        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `var-order`
            - Type: enum (`var-order`, `representations` or `both`)
            - Constraints: none
    - Examples: `force`, `force(metric=total-span)`, `force(metric=total-span, relations=configured, effect=both)`
- *Sliding window*

  Applies the sliding window algorithm.

    - Name: `slidwin`
    - Arguments:
        - `size`
            - Description: the maximum size of the window

        - Mandatory: no
        - Default: 4 (as configured by the *BDD sliding window size* option)
        - Type: number
        - Constraints: size must be in the range [1 .. 12]
      - `metric`
        - Description: the metric to use to compare orders
        - Mandatory: no
        - Default: `total-span`
        - Type: enum (`total-span` or `wes`)
        - Constraints: none
      - `relations`
        - Description: the kind of relations to use when computing metric values
        - Mandatory: no
        - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
        - Type: enum (`configured`, `legacy` or `linearized`)
        - Constraints: none
      - `effect`
        - Description: the effect of the variable orderer
        - Mandatory: no
        - Default: `var-order`
        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
    - Examples: `slidwin`, `slidwin(size=5)`, `slidwin(size=5, metric=total-span, relations=configured, effect=both)`
- *Sloan*

  Applies the Sloan profile/wavefront-reducing algorithm.

    - Name: `sloan`
    - Arguments:
      - `relations`
        - Description: the kind of relations to use when computing metric values
        - Mandatory: no
        - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
        - Type: enum (`configured`, `legacy` or `linearized`)
        - Constraints: none

        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `var-order`
            - Type: enum (`var-order`, `representations` or `both`)
            - Constraints: none
    - Examples: `sloan`, `sloan(relations=configured, effect=both)`
- *Weighted Cuthill-McKee*

  Applies the Weighted Cuthill-McKee bandwidth-reducing algorithm.

    - Name: `weighted-cm`
    - Arguments:
        - `node-finder`
            - Description: the node finder algorithm to use
            - Mandatory: no
            - Default: `george-liu`
            - Type: enum (`george-liu` or `sloan`)
            - Constraints: none
        - `relations`
            - Description: the kind of relations to use when computing metric values
            - Mandatory: no
            - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
            - Type: enum (`configured`, `legacy` or `linearized`)
            - Constraints: none
        - `effect`
            - Description: the effect of the variable orderer
            - Mandatory: no
            - Default: `var-order`
            - Type: enum (`var-order`, `representations` or `both`)
            - Constraints: none
    - Examples: `weighted-cm`, `weighted-cm(relations=configured)`, `weighted-cm(node-finder=george-liu, relations=configured, effect=both)`
- *Choice*

  Applies multiple algorithms to the same variable order and chooses the best resulting order.

    - Name: `or`

- Arguments:
  - `choices`
    - Description: the orderers to apply
    - Mandatory: yes
    - Default: n/a
    - Type: list of orderers
    - Constraints: at least two orderers must be specified
  - `metric`
    - Description: the metric to use to compare orders
    - Mandatory: no
    - Default: `wes`
    - Type: enum (`total-span` or `wes`)
    - Constraints: none
  - `relations`
    - Description: the kind of relations to use when computing metric values
    - Mandatory: no
    - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
    - Type: enum (`configured`, `legacy` or `linearized`)
    - Constraints: none
  - `effect`
    - Description: the effect of the variable orderer
    - Mandatory: no
    - Default: `var-order`
    - Type: enum (`var-order`, `representations` or `both`)
    - Constraints: none
- Examples: `or(choices=[dcsh,force])`, `or(choices=[(force -> dcsh), (dcsh -> force)], metric=wes, relations=configured, effect=both)`

- *Repeat*

  Repeat an orderer a number of times.

  - Name: `repeat`
  - Arguments:
    - `orderer`
      - Description: the variable orderer to repeatedly apply
      - Mandatory: yes

        - Default: n/a
        - Type: orderer
        - Constraints: none
    - `count`
        - Description: the number of types to apply the given variable orderer
        - Mandatory: yes
        - Default: n/a
        - Type: number
        - Constraints: count must be at least one
    - `intermediate-effect`
        - Description: the effect of the given variable orderer, for all but the final time it is applied
        - Mandatory: no
        - Default: `var-order`
        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
    - `final-effect`
        - Description: the effect of the given variable orderer, for the final time it is applied
        - Mandatory: no
        - Default: `var-order`
        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
  - Examples: `repeat(orderer=(dcsh->force->slidwin), count=4)`, `repeat(orderer=dcsh, count=2, intermediate-effect=both, final-effect=var-order)`
- *Reverse*

  Reverses the variable order.

  - Name: `reverse`
  - Arguments:
    - `relations`
        - Description: the kind of relations to use when computing metric values
        - Mandatory: no
        - Default: `configured` (per the *BDD hyper-edge creation algorithm* option)
        - Type: enum (`configured`, `legacy` or `linearized`)
        - Constraints: none
    - `effect`

        - Description: the effect of the variable orderer
        - Mandatory: no
        - Default: `var-order`
        - Type: enum (`var-order`, `representations` or `both`)
        - Constraints: none
    - Examples: `reverse`, `reverse(relations=configured, effect=both)`

Node finders

Orderers that work on graphs may use a pseudo-peripheral node finder algorithm while computing a variable order. Multiple such node finders can be used:

- *George-Liu*

  Use the algorithm by George and Liu.

  Name: `george-liu`

- *Sloan*

  Use the algorithm by Sloan.

  Name: `sloan`

Metrics

To compare different orders, and choose the best one, metric values are used. Multiple metrics can be used:

- *Total span*

  Use the total span metric.

  Name: `total-span`

- *WES*

  Use the Weighted Event Span (WES) metric.

  Name: `wes`

Relations

Metric value can be computed using different variable relations derived from the CIF specification. Multiple metrics can be used:

- *Legacy*

  Use the legacy relations.

Name: `legacy`

- *Linearized*

  Use the linearized relations.

  Name: `linearized`

- *Configured*

  Use the relations as configured using the *BDD hyper-edge creation algorithm* option.

  Name: `configured`

Effects

A variable orderer produces a variable order. The *effect* of a variable orderer determines what happens with the produced order:

- It may be used as the new variable order. The produced variable order then becomes the variable order to be used as input to the next orderer, or the final order in case it is the effect of the last orderer. If the produced variable order is not used as the new variable order, the variable order remains unchanged by the orderer.
- It may be used to compute new representations of the variable relations. Subsequent orderers will then use these new representations as input to algorithms and for computing metrics. If the produced variable order is not used to create new representations, the representations remain unchanged by the orderer.

The following effects can be configured:

- *Variable order*

  Uses the produced variable order as the new variable order. Does not update the representations.

  Name: `var-order`

- *Representations*

  Updates the representations. Does not use the produced variable order as the new variable order.

  Name: `representations`

- *Both*

  Uses the produced variable order as the new variable order, and updates the representations.

  Name: `both`

BDD operation cache

One of the main properties of the BDDs used by the data-based synthesis algorithm is that they employ full sharing. That is, if a part of a binary tree needs to be represented more than once, it is stored only once, and reused. This leads to BDDs being represented using directed acyclic graphs, rather than binary trees.

The BDD library uses an operation cache to speed up synthesis. Whenever a certain operation is performed on one or more nodes of a BDD graph, the result is cached. If that same operation is performed again on the same nodes, the cached result is reused, if still available. This way, repeated calculations can be prevented for shared sub-graphs.

The operation cache is essential for the performance of the synthesis algorithm. With an infinite cache, the operations are generally linear in the number of nodes used to represent the BDDs on which they are applied. Without caching, the computation time grows exponentially.

Obviously, in practice we can't have an infinite cache, as a computer only has a finite amount of memory available. We thus need to work with a finite cache. Whenever a new operation result doesn't fit in the cache, an older result is overwritten, and will need to be recomputed if it is needed again.

Increasing the cache size can significantly increase performance for large systems, as a cache that is too small is ineffective, and results in many operations needing to be repeated, that could have otherwise been obtained from the cache. However, a larger than needed cache may also significantly decrease performance, as a cache that is too large may no longer fit within CPU caches, leading to more expensive accesses to the main memory.

Several different operation caches are used during synthesis, to cache the results of various different types of operations. Note that the combination of these caches may grow quite large, even larger than the representations of the BDDs themselves, if too large cache sizes are used. Too large cache sizes are a common cause of out-of-memory errors.

All operation caches have the same size. Their size can be configured in two ways:

- *Fixed cache size*: If a single fixed cache size is used, the caches remain the same size during the entire synthesis. To configure a fixed cache size, use the *BDD library operation cache size* option.
- *Variable cache size*: If a variable cache size is used, the caches grow in size as the BDD node table grows in size. The ratio between the sizes of the caches and the node table can be configured using the *BDD library operation cache ratio* option. A variable cache size is used only if no fixed cache size is used.

State requirement invariants

Before synthesis, the entire CIF specification is converted to Binary Decision Diagrams (BDDs), a representation of predicates that allows for efficient synthesis. The state requirement invariants of the specification thus also need to be part of the BDD representation of the specification. These requirements can be enforced by applying them to the BDDs in multiple ways:

1. *Controlled behavior*:

All state requirement invariants are enforced via the 'controlled behavior' BDD. This BDD represents all states of the controlled system. Since the controlled system has to adhere to the state requirement invariants, these invariants can be put directly into the controlled behavior BDD.

2. *Edge guard or controlled behavior*:

   The decision how to enforce state requirement invariants is made per edge. For each edge with a controllable events, the invariants are enforced via the guards of the edge, by strengthened the existing guards with sufficient extra conditions that ensure that after taking the edge the state requirement invariants hold. The initialization predicate of the controlled system is restricted as well, to ensure the state requirement invariants also hold in initial states. As synthesis may not restrict the guards of edges with uncontrollable events, for those edges the extra restrictions are put into the controlled behavior BDD instead.

By default, all state requirement invariants are enforced via the controlled behavior. This can be changed by using the *State requirement invariant enforcement* option (see the options section).

The effect of the different approaches on the computational effort of synthesis can be quite complex:

- *Upfront effort*: Using approach 2, the edge guard restrictions need to be computed for each edge and each state requirement invariant, which potentially requires significant computational effort. Using approach 1, each state requirement invariant is considered only once, when putting it into the controlled behavior, which may require less computational effort.
- *Size of controlled behavior*: Reachability computations rely heavily on the controlled behavior predicate. Using approach 1, putting state requirement invariants into the controlled behavior, may restrict the controlled behavior, potentially leading to a smaller predicate and reduced computational effort. However, putting state requirement invariants into the controlled behavior may also lead to a controlled behavior predicate with more BDD nodes to express the additional restrictions, potentially increasing the computational effort.
- *Reachability fixed point*: Using approach 1, with a more restricted controlled behavior predicate, may lead to more quickly reaching a fixed point in reachability computations, potentially reducing computational effort.
- *Post-synthesis effort*: Using approach 1, the state requirement invariants are put into the controlled behavior, but they still need to be put into the edge guards after synthesis, which requires computational effort. Using approach 2, putting the state requirement invariants into the edge guards at the start of synthesis, means that the restrictions do not need to be applied at the end of synthesis, potentially reducing computational effort.

Both approaches have potential benefits and drawbacks, making for a trade-off between their various effects. Which approach is most efficient depends on the model.

For more information, see the scientific publication by [Thuijsman et al. (2021)], where the two approaches are compared.

Debug output

By default, the data-based synthesis algorithm shows no progress information, and does not explain how the resulting supervisor is obtained. By enabling debug output, detailed information is printed to the console. Debug output can be enabled by setting the *Output mode* option (*General* category) to *Debug*.

Bounded reachability requirements

For bounded reachability requirements, debug output also reports the configured bound, the minimum-bound status and the reason that the bounded computation stopped. These messages describe an individual backward reachability computation within the overall synthesis computation. Synthesis can repeat this computation as other requirements and computations restrict the controlled behavior, so intermediate results may differ from those for the final supervisor.

For a requirement with `from`, the cover states are the source states to which the requirement currently applies, taking the current controlled behavior and any `avoid` or `stay` restriction into account. Cover states are covered when the computation has found a path to the goal from every such source state. Different source states may use different paths, possibly to different goal states, and their shortest permitted paths may have different lengths. If there are applicable source states and all are covered, the minimum bound is the largest of these shortest-path lengths: it is the smallest bound sufficient for every applicable source state.

For example, a bounded reachability requirement with `from: source` may have `bound: 10`, while one source state needs only one transition to reach the goal and another needs three. If all other applicable source states can also reach the goal within three transitions, debug output contains:

```
... [bounded result, bound: 10, minimum bound: 3, cover states covered].
```

Coverage can also hold at depth zero, either because all applicable source states already satisfy the goal or because there are no applicable source states. Thus, coverage alone does not show that source states remain reachable from an initial state.

If a fixed point is reached first, the reported minimum bound instead indicates the depth beyond which no additional states are found. For example:

```
... [bounded result, bound: 10, minimum bound: 2, fixed point].
```

This means that allowing more than two transitions would not add any states to this reachability result. It does not imply that all applicable source states can reach the goal: some may be unable to do so under the current restrictions, regardless of the bound. Without `from`, there are no cover states to check, and a minimum bound is reported only if a fixed point is reached.

If the configured bound is exhausted before either condition is reached, debug output instead contains, for example:

```
... [bounded result, bound: 10, minimum bound: not determined, bound reached].
```

For a requirement with `from`, this means that not all source states to which the requirement currently applies could reach the goal within the configured bound.

Performance implications

Enabling debug output may significantly slow down the synthesis algorithm, especially for larger models. The performance degradation stems mostly from the printing of predicates. Predicates are internally represented using Binary Decision Diagrams (BDDs). To print them, they are converted to CNF or DNF predicates, similar to one of the approaches to convert BDDs to CIF predicates for synthesis output.

To limit the performance degradation, options are available to limit the conversion of BDDs to CNF/DNF predicates. The *BDD debug max nodes* controls the maximum number of BDD nodes for which to convert a BDD to a readable CNF/DNF representation for the debug output. The default is `10` nodes. The maximum must be in the range [1 .. 2^{31} - 1]. The option can be set to have an infinite maximum (no maximum), using option value `inf`. The *BDD debug max paths* option controls the maximum number of BDD true paths for which to convert a BDD to a readable CNF/DNF representation for the debug output. The default is `10` paths. The maximum must be non-negative. The option can be set to have an infinite maximum (no maximum), using option value `inf`. If a BDD has more than the specified maximum number of nodes, or more than the specified number of true paths, it is not converted to a CNF/DNF predicate. Instead, it is converted to a textual representation that indicates the number of nodes and true paths, e.g. `<bdd 1,234n 5,678p>` for a BDD with 1,234 nodes and 5,678 true paths.

By limiting the conversion of BDDs to CNF/DNF predicates, debug output can still be used for large models to see progress information, while not degrading the performance too much.

Statistics

The data-based synthesis tool supports printing various kinds of statistics. By default, no statistics are printed. Statistics can be enabled using the *Statistics* option (see the options section). The following statistics are available:

- *BDD garbage collection* [`bdd-gc-collect`]

  Prints BDD garbage collection information before and after each garbage collection.

  The BDD library that is used maintains an internal node table. Whenever new nodes no longer fit in the node table, a garbage collection is started by the BDD library, to clean out all nodes that are no longer needed. See also the BDD library initial node table size option (see the options section).

  BDD garbage collection information is printed to the console before and after each garbage collection. The printed information includes the size of the node table, the number of free nodes, timing information, etc.

- *BDD node table resize* [`bdd-gc-resize`]

  Prints BDD node table resize information each time a BDD node table resize is performed.

  The BDD library that is used maintains an internal node table. Whenever new nodes no longer fit in the node table, a garbage collection is started by the BDD library, to clean out all nodes that are no longer needed. If only very limited space could be reclaimed by garbage collection, the size of the node table is increased. See also the BDD library initial node table size option (see the options section).

  BDD node table resize information is printed to the console each time the node table is resized. The printed information includes the old and new sizes of the node table.

- *BDD node and operation* [`bdd-perf-node-op`]

  Prints metrics related to the BDD nodes and operations.

  The BDD library that is used maintains an internal node table to store the nodes that comprise the various BDDs. If a node already exists in the node table, it is reused rather than creating a new one, to ensure each node exists only once in the node table. The library will use the node's hash to determine where to find or store it in the node table. Due to hash collisions, the library may not be able to find or store the node at the entry corresponding with the hash. It will then consider subsequent entries in the node table.

  The BDD library may perform operations on BDD nodes. An operation may operate on a single BDD, such as the the logical negation (`not`) operation. It may also operate on multiple BDDs, such as the logical conjunction (`and`) operation. Regardless, BDD operations are generally implemented recursively on their structures. They start at the root nodes, go further and further towards the terminal nodes, and recursively consider all relevant nodes as they encounter them. The BDD library uses caches to store previously computed results of operations, to prevent having to recalculate them time and again, speeding up calculations.

  For more information, see also the section about BDD operation caches.

  The BDD node and operation information is printed to the console, after execution, just before termination of the tool. The table below shows all metrics that are printed and their meaning.

  Note that collecting these metrics makes synthesis take longer than without collecting it.

Metric	Description
Node creation requests	The number of times a node creation is requested, regardless of whether this node is already present in the node table or not. This count includes nodes that are never stored, such as nodes with the same low and high children. No node creation requests are ever issued for the terminal nodes ('true' and 'false').
Node creation hits	The number of times a node creation is requested, while the node is already in the node table. This count excludes nodes that are never stored, such as nodes with the same low and high children.

Metric	Description
Node creation misses	The number of times a node creation is requested, while the node is not already in the node table, and thus the node needs to be created and added to the node table. This count excludes nodes that are never stored, such as nodes with the same low and high children.
Node creation chain accesses	The number of extra node table entries that are checked in case of hash collisions. This count represents the number of times that an extra (non-first) entry of the node table is considered to find and/or store the node. If for a single node creation multiple extra entries are checked, each of these extra entries is counted separately. This count excludes nodes that are never stored, such as nodes with the same low and high children.
Operation count	The number of times a BDD operation is performed on one or more nodes. This includes not only root nodes of BDDs, but also all recursive operations for parts of the BDDs. This includes the trivial cases where the cache is not checked, e.g., for `not true`, `x and x`, `x and true`, `false and x`, etc.
Operation cache hits	The number of times a BDD operation is performed on one or more nodes, and the result is already in the cache. This includes not only root nodes of BDDs, but also all recursive operations for parts of the BDDs. This excludes the trivial cases where the cache is not checked.
Operation cache misses	The number of times a BDD operation is performed on one or more nodes, and the result is not already in the cache. This includes not only root nodes of BDDs, but also all recursive operations for parts of the BDDs. This excludes the trivial cases where the cache is not checked.

- *Continuous BDD performance statistics* [`bdd-perf-cont`]

  Prints continuously collected platform and machine independent BDD performance related metrics.

  This statistic continuously collects the platform and machine independent BDD performance metrics described in the table below. These metrics are discussed in more detail in [Thuijsman et al. (2019)].

  The information is printed to a CSV file that can be configured using the *Continuous performance statistics file* option. The file is written after execution, just before termination of the tool.

  Note that collecting these metrics makes synthesis take longer than without collecting it.

Metric	Description
Operations	The number of BDD operations performed on BDD nodes so far during synthesis, for which the cache is checked and the result is not already in the cache. The number of operations performed on BDD nodes is a platform and machine independent measure of the amount of time needed to perform BDD operations, due to BDD operations being implemented recursively on their node graphs.
Used BBD nodes	The number of BDD nodes currently in use to represent all BDDs. The size of a BDD, expressed as the number of nodes used, is a platform and machine independent measure of the amount of memory needed to store it.

- *Controlled system states* [`ctrl-sys-states`]

  Prints the number of states in the resulting controlled system (the uncontrolled system restricted by the synthesized supervisor).

  If the resulting supervisor (and thus the controlled system) is empty, or if forward reachability is enabled, the number of states that is printed is an exact number (e.g. `exactly 0 states`, `exactly 1 state`, `exactly 1,234 states`). In other situations, the controlled behavior predicate that is used to determine the number potentially gives an over-approximation, and an upper bound on the number of states is printed (e.g. `at most 1,234 states`).

  This metric is printed to the console, after the supervisor is computed, and before it is simplified and written to a file.

- *Maximum used BDD nodes* [`bdd-perf-max-nodes`]

  Prints the maximum number of BDD nodes used during synthesis.

  The size of a BDD, expressed as the number of nodes used, is a platform and machine independent measure of the amount of memory needed to store it. This metric is discussed in more detail in [Thuijsman et al. (2019)].

  This metric is printed to the console, after execution, just before termination of the tool. It prints the maximum number of BDD nodes used during synthesis.

  Note that collecting this metric makes synthesis take considerably longer than without collecting it.

- *Timing* [`timing`]

  Print information for timing of various parts of the tool.

  Timing is only collected for parts of the tool that were actually executed. Timing is represented as a tree. The root of the tree represents the total time of the synthesis tool. For some parts of the tool, timing is also collected for sub-parts.

  Timing information is printed to the console, after execution, just before termination of the tool. Durations are all printed in milliseconds, to make it easier to compare timing for various parts.

When measuring timing performance, always perform multiple measurements, and take the average. Also, use a warm-up phase, to avoid skewed results.

- *Maximum used memory* [`max-memory`]

  Prints the maximum memory used during synthesis.

  Measuring the memory usage may be useful, for instance to see how close you are to memory limits. It may especially be useful if you want to subsequently extend your model, making it more complex and potentially requiring more memory for synthesis.

  This metric is printed to the console, after execution, just before termination of the tool. It prints the maximum amount of memory used during synthesis, in number of bytes and in a more readable representation such as mebibytes or gibibytes.

  Note that collecting this metric may make synthesis take longer than without collecting it.

  The CIF data-based synthesis tool is written in Java, a managed language with a garbage collector. Hence, the measured memory usage is a best-effort approximation, and may not be correct.

  When measuring memory usage, always perform multiple measurements, and take the average.

  If memory usage is measured using the Eclipse ESCET IDE, make sure not to perform multiple syntheses in parallel, and make sure the Eclipse ESCET IDE is not performing any other operations. However, when measuring the memory usage in the Eclipse ESCET IDE, the IDE itself, as well as editor content, console content, and so on, are also included in the memory usage. It is therefore highly recommended to use the Eclipse ESCET command line scripts rather than the Eclipse ESCET IDE to perform memory usage measurements.

  To ensure the best result, let Java perform a garbage collection before starting the data-based synthesis tool. This ensures that synthesis starts with 'clean' memory. This is particularly relevant if multiple syntheses are performed one after the other, to ensure that any previous syntheses do not affect subsequent measurements. Both synthesis and garbage collection may conveniently be executed via a ToolDef script. For example, using a ToolDef script like this:

  ```
  from "lib:cif" import *;
  import java.lang.System.gc;
  gc();
  cifdatasynth(...);
  ```

  The maximum used memory statistics obtained in this way are an easy way to get an approximation of the memory used during synthesis. For a more comprehensive way to measure memory usage, use a tool like VisualVM.

In the option dialog, each of the different kinds of statistics can be enabled and disabled individually, using a checkbox.

From the command line, using the `--stats` option, the names of the different kinds of statistics, as indicated above between square brackets, should be used, separated by commas. For instance, use

`--stats=bdd-gc-collect,bdd-gc-resize` to enable both BDD garbage collection statistics and BDD node table resize statistics, but keep all other statistics disabled.

Specifying a statistics kind twice leads to a warning being printed to the console.

References

- [Aloul et al. (2003)] Fadi A. Aloul, Igor L. Markov and Karem A. Sakallah, "FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic", In: Proceedings of the 13th ACM Great Lakes symposium on VLSI, pages 116-119, 2003, doi:10.1145/764808.764839
- [George and Liu (1979)] Alan George and Joseph W. H. Liu, "An Implementation of a Pseudoperipheral Node Finder", ACM Transactions on Mathematical Software, volume 5, issue 3, pages 284-295, 1979, doi:10.1145/355841.355845
- [Hendriks et al. (2025)] Dennis Hendriks, Michel Reniers, Wan Fokkink and Wytse Oortwijn, "Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0", arXiv, 2025, doi:10.48550/arXiv.2511.04370
- [Lousberg et al. (2020)] Sam Lousberg, Sander Thuijsman and Michel Reniers, "DSM-based variable ordering heuristic for reduced computational effort of symbolic supervisor synthesis", IFAC-PapersOnLine, volume 53, issue 4, pages 429-436, 2020, doi:10.1016/j.ifacol.2021.04.058
- [Minato (1996)] Shin-ichi Minato, "Binary Decision Diagrams and Applications for VLSI CAD", The Kluwer International Series in Engineering and Computer Science, volume 342, Springer, 1996, doi:10.1007/978-1-4613-1303-8
- [Ouedraogo et al. (2011)] Lucien Ouedraogo, Ratnesh Kumar, Robi Malik and Knut Åkesson, "Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata", IEEE Transactions on Automation Science and Engineering, volume 8, issue 3, pages 560-569, 2011, doi:10.1109/TASE.2011.2124457
- [Sloan (1989)] S. W. Sloan, "A FORTRAN program for profile and wavefront reduction", International Journal for Numerical Methods in Engineering, volume 28, issue 11, pages 2651-2679, 1989, doi:10.1002/nme.1620281111
- [Thuijsman et al. (2019)] Sander Thuijsman, Dennis Hendriks, Rolf Theunissen, Michel Reniers and Ramon Schiffelers, "Computational Effort of BDD-Based Supervisor Synthesis of Extended Finite Automata", In: Proceedings of the IEEE Conference on Automation Science and Engineering, pages 486-493, 2019, doi:10.1109/COASE.2019.8843327
- [Thuijsman et al. (2021)] Sander Thuijsman, Michel Reniers and Dennis Hendriks, "Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis", In: Proceedings of the 2021 IEEE 17th International Conference on Automation Science and Engineering, pages 777-783, 2021, doi:10.1109/CASE49439.2021.9551593

4.3.2. CIF multi-level splitter

The CIF multi-level splitter is used to split one CIF specification into several smaller co-operating CIF specifications. The behavior of the specified system does not change. If you synchronize the smaller specifications, for example by merging them again, they behave the same as the original

specification.

One use of this application is to enable data-based synthesis on large specifications. Synthesizing a large specification directly with data-based synthesis may take a very long time, or synthesis may not have enough memory available to complete its task. By first splitting the specification, performing synthesis on each part and finally merging the results again, you may be able to avoid such problems.

A specification cannot be split in arbitrary parts. In order to preserve the behavior of the specification, some of the automata, input variables and requirements that share certain CIF elements with each other must always be kept together. For example, a requirement using the value of a discrete variable must stay together with that discrete variable, and thus with the automaton that owns the discrete variable.

After creating clusters of such CIF elements, you can perform synthesis on each cluster and construct a local supervisor for it. By taking all local supervisors together, the supervised behavior of the complete specification is obtained. That behavior is the same as the behavior of a supervisor derived from the complete original specification, except that taking the supervisors of all partial specifications together may be blocking. That is, the non-blockingness guarantee still holds for each local supervisor, but it is no longer guaranteed for the complete specification.

In principle, each requirement could be put in a cluster together with its related plants. However, this may lead to a large number of clusters, and a lot of duplicated synthesis work. For instance, if two requirements relate to the same plants, it is more efficient to cluster the two requirements together with those shared plants. On the other hand, clustering everything into a single cluster returns us to the original scalability issues of synthesis on a single, large specification. It is therefore essential to strike a good balance between splitting the large specification into parts (to improve performance) and not splitting too much (to prevent duplicate work).

The multi-level splitter helps with this by automatically clustering the elements of the specification. It keeps together elements that must be kept together, and furthermore tries to find a good clustering that will give good synthesis performance. Intuitively, it clusters strongly related parts of the specification, while keeping less related parts in separate clusters. More concretely, it first finds plant groups that consist of plant elements that must be kept together, and requirement groups that consist of requirement elements that must be kept together. It then relates the plants and requirement groups, putting them into a Domain Mapping Matrix (DMM). For more detailed information about these relations, plant and requirement groups, and the construction of the DMM, see the CIF relations collector's output section. From the DMM, a Design Structure Matrix (DSM) is created, that relates plant groups by shared requirements. A DSM clustering algorithm is then used to create a multi-level clustering of the plant groups. The algorithm is heuristic in nature, but generally gives good results, and can be tuned using various settings if needed.

The multi-level tool does not perform synthesis. Instead, it generates a tree of partial specifications, one per cluster in the multi-level clustering. Each partial specification contains the relevant plants and requirements for that cluster. As a service for further processing of the partial specifications, the multi-level splitter also checks whether each partial specification complies with the pre-conditions of data-based synthesis, and it generates a ToolDef script to reduce the effort to synthesize a supervisor from each partial specification.

By using the generated ToolDef script you can synthesize the partial specifications. If synthesis is still infeasible with the initial configuration, the generated script gives room to tweak synthesis settings for each partial specification separately.

The multi-level splitter is based on the approach outlined in [Goorden et al. (2020)].

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Apply multi-level splitting...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Apply multi-level splitting...**.
- Use the `cifmultilevel` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifmultilevel` command-line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative file system path to the input CIF file.
- *DMM output file*: The absolute or relative file system path to which to write information about the generated DMM. This file can be used for further analysis. When no path is specified for the option, the DMM information is not written.
- *Partial specifications output directory*: The absolute or relative path to the directory to write all output files. If no path is specified, a path is constructed from the name of the input specification by removing the `.cif` extension if available, and appending `_multilevel`. The tool generates several files in the directory: a CIF file for each partial specification, and a ToolDef script to perform data-based synthesis on each of these partial specifications.

Clustering options

The clustering algorithm is heuristic in nature. While theoretically each result is good, practically it may not always suit your purpose. For instance, the automatic clustering could in certain cases result in partial specifications that do not provide good synthesis performance.

The following options allow tuning the resulting split:

- *Evaporation factor*: Factor that influences when a node is considered to be part of a cluster. Higher values leads to fewer nodes in a single cluster and thus more (smaller) clusters. The value must be at least `1.0` and at most `10.0`. Usually it is between `1.5` and `3.5`. The default value is `2.0`.

- *Inflation factor*: Factor that influences how fast large values increase and small values decrease, where the small values are eventually eliminated. Higher values of the factor speed up the process. The value must be at least `1.0` and at most `4.0`. Usually it is between `1.5` and `3.5`. The default value is `2.0`.
- *Bus detection algorithm*: The bus detection algorithm to apply. By default, no bus is detected.
- *Bus factor*: Factor that influences when a node is considered to be part of the bus. The actual interpretation of this factor depends on the chosen bus detection algorithm. The default value is `2`.
- *Convergence limit*: Allowed remaining numerical error before considering termination of the algorithm. Higher values end the computation sooner at the cost of less precision in the results. The value must be between `0` and `1` (where `0` is not achievable, and `1` is not precise enough). The default value is `1e-4`.
- *Step count*: Number of additional nodes to visit each iteration. The value must be at least `1` and at most `4`. The default value is `2`. Changing this values is rarely needed.

More details about the clustering algorithm are available as part of the DSM clustering tool documentation.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove annotations (removes all annotations, except reachability requirement annotations).
- Eliminate component definition/instantiation
- Eliminate automaton `self` references
- Remove I/O declarations
- Simplify values

Supported specifications

The CIF multi-level splitter tool has performed pre-processing before performing the checks below, which means that:

- All annotations are ignored, except reachability requirement annotations.
- I/O declarations are ignored. A warning is printed if the specification contains a CIF/SVG input mapping.
- Component definitions and instantiates are allowed (they have been eliminated already).
- Automata references with `self` are allowed (they have been rewritten).

The CIF multi-level splitter allows a subset of CIF specifications. Many of the restrictions are inherited from the data-based synthesis application. It may be a good idea to consult its

documentation of supported specifications when in doubt if something is supported. Note that the CIF multi-level splitter tool checks the data-based synthesis pre-conditions on its generated partial output specifications in an effort to catch problems as early as possible.

- Regarding automata and invariants:
  - Automata must have exactly one initial location, and it must be possible to determine this statically.
  - Initialization predicates in components, so outside of locations, are not allowed.
  - Marker predicates in locations are only allowed if they can be statically evaluated to either `true` or `false`.
  - Marker predicates in components, so outside of locations, are not allowed.
  - Only plant and requirement automata are allowed. Kindless and supervisor automata are not allowed.
    - Only requirement invariants are allowed. Kindsless, plant and supervisor invariants are not allowed.
  - The specification must have at least one input variable or one plant automaton. (An input variable counts as plant in this case.)
  - The specification must have at least one requirement automaton or one requirement invariant.
  - State requirement invariants are not allowed. Only state/event exclusion requirement invariants are allowed.
  - Invariants in locations are not allowed. Only invariants in components are allowed.
- Regarding events:
  - Events must be either controllable or uncontrollable.
  - Tau events are not allowed.
  - Channels (events with data types) are not allowed.
  - Updates must be simple assignments. Conditional updates (`if` updates), multi-assignments, and partial assignments are not allowed.
- Regarding variables:
  - Discrete variables must have one initial value that can be statically computed.
  - Continuous variables are not allowed.
  - Equations are not allowed.
  - Functions in any form are not allowed. This includes both standard library functions and user-defined functions.
- Regarding data types:
  - The boolean type is allowed.
  - Enumeration types are allowed.
  - Ranged integer types are allowed.

    - All other types are not allowed.
- Regarding expressions, only operators that take and produce data with a supported type are allowed. For example, regular division is not allowed since it produces a `real` value, and casting from a `string` to an integer is not allowed since the `string` type is not allowed.
    - The data-based synthesis tool adds additional restrictions beyond the type of the data. Consult the elaborate description on supported specifications for data-based synthesis for details.
- Furthermore:
    - Reachability requirement annotations are not allowed.

Output of the CIF multi-level splitter

As discussed in the general introduction above, the multi-level splitter constructs several partial specifications from a single CIF specification. Each partial specification corresponds to a cluster in the multi-level clustering, and contains a part of the original specification. The partial specifications thus also form a tree.

All partial specifications for the plant groups are at the bottom of the tree, in the leaf nodes. All requirements that only relate to a single plant group are positioned in the same cluster as the corresponding plant group. Each level up in the tree contains clusters and their related partial specifications that cover more global parts of the system. For instance, consider a cluster at the second level that has two children. Then it contains the two plant groups of those children, but also all requirements that only relate to those two children and that weren't already included at a lower level. All the partial specifications together cover all the plants and requirements of the entire original specification.

Partial specifications do no import their child partial specifications. Instead, their name indicates their place in the tree. The `partialSpec.cif` specification is at the top (highest level) of the tree. All other specifications have an index number for each level appended in their name. For example `partialSpec_2_3.cif` means that this file is the third part of the second part of the entire specification.

In addition, the multi-level splitter assists in performing data-based synthesis on the partial specifications. First it checks each partial specification against the pre-conditions of synthesis and reports the findings in a brief manner.

It also generates a ToolDef script to perform data-based synthesis on each of the partial specifications. Inside the script, the partial specifications with problems with respect to the data-based synthesis tool's pre-conditions are commented out. The problem reports are also included as comment in the script for your convenience. In this way, you can quickly get an overview of the found problems and decide how to proceed. For example, you may want to change the original specification and perform multi-level splitting again.

If you decide to proceed by performing data-based synthesis on the partial specifications, the generated script has room for customizing data-based synthesis settings for each of the partial specifications individually.

Example

Suppose that in a system there are two counters, and the first counter may not increase if it would get bigger than the second counter. The specification is:

```
plant def Counter():
  controllable c_up, c_down;
  disc int[1..10] counter = 1;

  location:
    initial;
    edge c_up do counter := counter + 1;
    edge c_down do counter := counter - 1;
end

first: Counter();
second: Counter();

requirement first.first_up needs second.counter > first.counter;
```

Splitting this specification with the multi-level splitter reports:

```
Partial specification "partialSpec.cif":
    Plants:
        - Plant "first".
        - Plant "second".

    Requirements:
        - Requirement "requirement invariant first.first_up needs second.counter >
first.counter".

    Children of the partial specification:
        - Partial specification "partialSpec_1.cif".
        - Partial specification "partialSpec_2.cif".

Partial specification "partialSpec_1.cif":
    Plants:
        - Plant "first".

    Requirements:
        None.

    Children of the partial specification:
        None.

Partial specification "partialSpec_2.cif":
    Plants:
        - Plant "second".
```

```
    Requirements:
        None.

    Children of the partial specification:
        None.
```

As you can see, there are two specifications at the bottom of the tree. They are `partialSpec_1.cif` and `partialSpec_2.cif`. Each contains one of the automata.

Above them is the top-level `partialSpec.cif` that contains the shared requirement. It also contains all parts of the specification needed to compute the requirement. For this small example, that is unfortunately the entire specification.

References

- [Goorden et al. (2020)] Martijn Goorden, Joanna van de Mortel-Fronczak, Michel Reniers, Wan Fokkink and Jacobus Rooda, "Structuring Multilevel Discrete-Event Systems With Dependence Structure Matrices", IEEE Transactions on Automatic Control, volume 65, issue 4, pages 1625-1639, 2020, doi:10.1109/TAC.2019.2928119

4.3.3. Event-based synthesis toolset

Synthesis, 'supervisor synthesis', or 'supervisory controller synthesis', is a generative technique, where one derives a (supervisor) automaton from a collection of plants and requirements. Synthesis is an essential part of the synthesis-based engineering approach to develop supervisory controllers. The resulting supervisor is maximally permissive under the conditions of being free of deadlocks, and always having the option of reaching a marked state.

While there is only one true synthesis tool (the tool that actually derives a supervisor automaton from a collection of plant and requirement automata), other tools exist to support the process. These tools together form the *event-based synthesis toolset*.

These tools use and modify the sequences of events that can be performed. This in contrast to state-based tools, which operate primarily on the state of the system. Event sequences directly hook into language theory, which places these tools firmly in the language theory mathematical framework.

When employing synthesis, consider also the data-based synthesis tool. It is often much more efficient when synthesizing supervisors for large and complex systems. It also supports a larger subset of CIF concepts.

Supported specifications

The application supports a subset of CIF specifications. The following restrictions apply:

- Channels (events with data types) are not supported.
- Any use of the `tau` event is not supported. Note that both explicit use (keyword `tau`) and implicit use (no events on an edge) are unsupported.

- Automata with multiple initial locations are not supported.
- Edges with updates are not supported.
- Urgent locations and edges are not supported.
- Initialization predicates in components are not supported.
- Initialization predicates in locations that are not trivially true or false are not supported.
- Marker predicates in components are not supported.
- Marker predicates in locations that are not trivially true or false are not supported.
- Invariants are not supported, unless they don't impose any restrictions.
- Guards of edges that are not trivially true or false are not supported.

To allow state/event exclusion invariants to be used in the input, manually eliminate them first using the Eliminate state/event exclusion invariants CIF to CIF transformation.

For meaningful results, it is recommended to have an initial location and at least one marked location, in each automaton.

I/O declarations are ignored. For tools that produce CIF models, those models will not contain any I/O declarations.

Annotations are ignored. For tools that produce CIF models, those models will not contain any annotations, except that they may contain state annotations.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Eliminate component definition/instantiation

Automaton kinds

The event-based toolset recognizes the CIF automaton kinds `plant`, `requirement`, and `supervisor`, and interprets them using the corresponding concepts of the mathematical framework of supervisor synthesis. Kindless/regular automata (without a supervisory kind) are treated as unknown.

Location names in reports and errors

Several tools output reports or give errors with locations that are interesting in some way.

Tools that take a single automaton as input report locations by their name in the input specification, for example `location "button.off"` denotes the `off` location in the `button` automaton. The location of automata that have a single unnamed location use `*` as location name, for example `location`

`"req.*"` denotes the single unnamed location in the `req` automaton.

Tools that take a number of automata together, like supervisor synthesis report interesting states as `state` followed by the names of the locations, for example `state "button.on", "machine.idle"`. Such a state refers to a location that represents the combined locations of the automata, in the example, the combined locations `button.off` and `machine.idle`.

Tools that combine or merge locations, like projection, report such combined locations as a partition, for example `partition "machine.down", "machine.off"` represents a location that is a combination of the `down` and the `off` locations in the `machine` automaton.

Available tools

- Event-based synchronous product

  Computes the product of a number of deterministic or non-deterministic automata, where common events are synchronized. Essentially computes the state space.

- Event-based supervisor synthesis

  Derivation of a maximal permissive supervisor from a collection of deterministic plant and requirement automata.

- Event-based synthesis analysis

  Analysis of the removed parts of a synthesized supervisor.

- Event-based nonconflicting check

  Verifies whether automata are conflicting, that is together could lead to non-coreachable states.

- Event-based controllability check

  Verifies whether the supervisor does not disable uncontrollable events of the plant.

- Event-based language equivalence check

  Verifies whether two automata produce the same events at each point.

- Event-based NFA to DFA automaton conversion

  Converts a non-deterministic automaton to a deterministic automaton while preserving the event behavior.

- Event-based DFA minimization

  Minimize the number of locations of an automaton while preserving the event behavior.

- Event-based automaton projection

  Computes a projection of an automaton, resulting in a deterministic and language equivalent

automaton over a subset of its alphabet.

- Event-based observer check

  Verifies whether an automaton can act as an observer of occurrences of observable events.

- Event-based automaton abstraction

  Abstracts an automaton to a set of observable events.

- Event-based trim check

  Verifies whether the automata are trim, that is, in each automaton, the locations must be both reachable and co-reachable.

- Event-based trim

  Removes all locations that are not reachable or coreachable.

Event-based synchronous product

The event-based synchronous product takes one or more plant or requirement automata, and computes the synchronous product. Synchronous product means events on edges can only be taken if all automata with that event in their alphabet can take an edge with the same event at that time. This tool essentially computes the state space.

The tool takes a CIF file containing at least one automaton, and produces a new CIF file with the product of the automata. In addition, the general event-based restrictions apply as well.

If the kinds of the source automata are all the same, the resulting product automaton is of that kinds as well. Otherwise, the product automaton has unknown kind.

Starting the synchronous product tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply synchronous product...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply synchronous product...**.
- Use the `cifprod` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifprod` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_product.cif` file extension is added. The `product` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the product automaton. If not specified, defaults to `product`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default. If the output CIF file has an automaton with only a single non-initial location, then the location does not get a state annotation, regardless of whether the option is enabled or not.

Event-based supervisor synthesis

The supervisor synthesis procedure takes one or more deterministic plant automata, zero or more deterministic requirement automata, and combines them to a maximal permissive supervisor.

The tool takes a CIF file containing `plant` and (optionally) `requirement` automata. Besides the general event-based restrictions, the tool does not support:

- Having no `plant` automata at all.
- Having a `supervisor` or kindless automaton in the file. See Automaton kinds for a list of all automaton kinds.
- Having any non-deterministic automata.
- Events in the alphabet of the requirements that are not in the alphabet of the plants.
- Events that are not controllable or uncontrollable.

In addition, it warns about common mistakes:

- Automata without a marked location.
- Non-trim automata.
- Specifications without any `requirement` automata.

Finally, it can also perform checks about correct constructs that may not be the intention of its author. Each of these checks has to be enabled with an option:

- Automata with a marked deadlock location. In a system with infinite behavior, you should probably never enter a deadlock location.
- Automata with an empty alphabet. Such automata never participate in an event, and can be removed.
- Controllable events that are used in exactly one automaton. Not always wrong, but creating a controllable event, and not using it for control may be a mistake.

- Groups of automata that share events only within the group (and not with any automaton outside the group). Such a group is completely independent, and can be synthesized separately.

The synthesis tool produces a new CIF file with the supervisor automaton (of kind `supervisor`) if the resulting automaton has at least an initial location. Otherwise, synthesis fails with a `Supervisor is empty` error, and no CIF file is written.

When employing synthesis, consider also the data-based synthesis tool. It is often much more efficient when synthesizing supervisors for large and complex systems. It also supports a larger subset of CIF concepts.

Starting the supervisor synthesis tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply supervisor synthesis...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply supervisor synthesis...**.
- Use the `cifsupsynth` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifsupsynth` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_sup.cif` file extension is added. The `sup` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the supervisor automaton. If not specified, defaults to `sup`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default.
- *Enable Synthesis dump*: If enabled, the synthesis algorithm writes a dump file containing the actions performed by the algorithm, for later analysis. The name of the file is decided by the *Dump file* option.
- *Synthesis dump file*: The name of the dump file. Setting this option will enable dumping of the synthesis algorithm actions. The options contains the absolute or relative local file system path to the synthesis dump file. If not specified, defaults to the input file path, where the `.cif` file

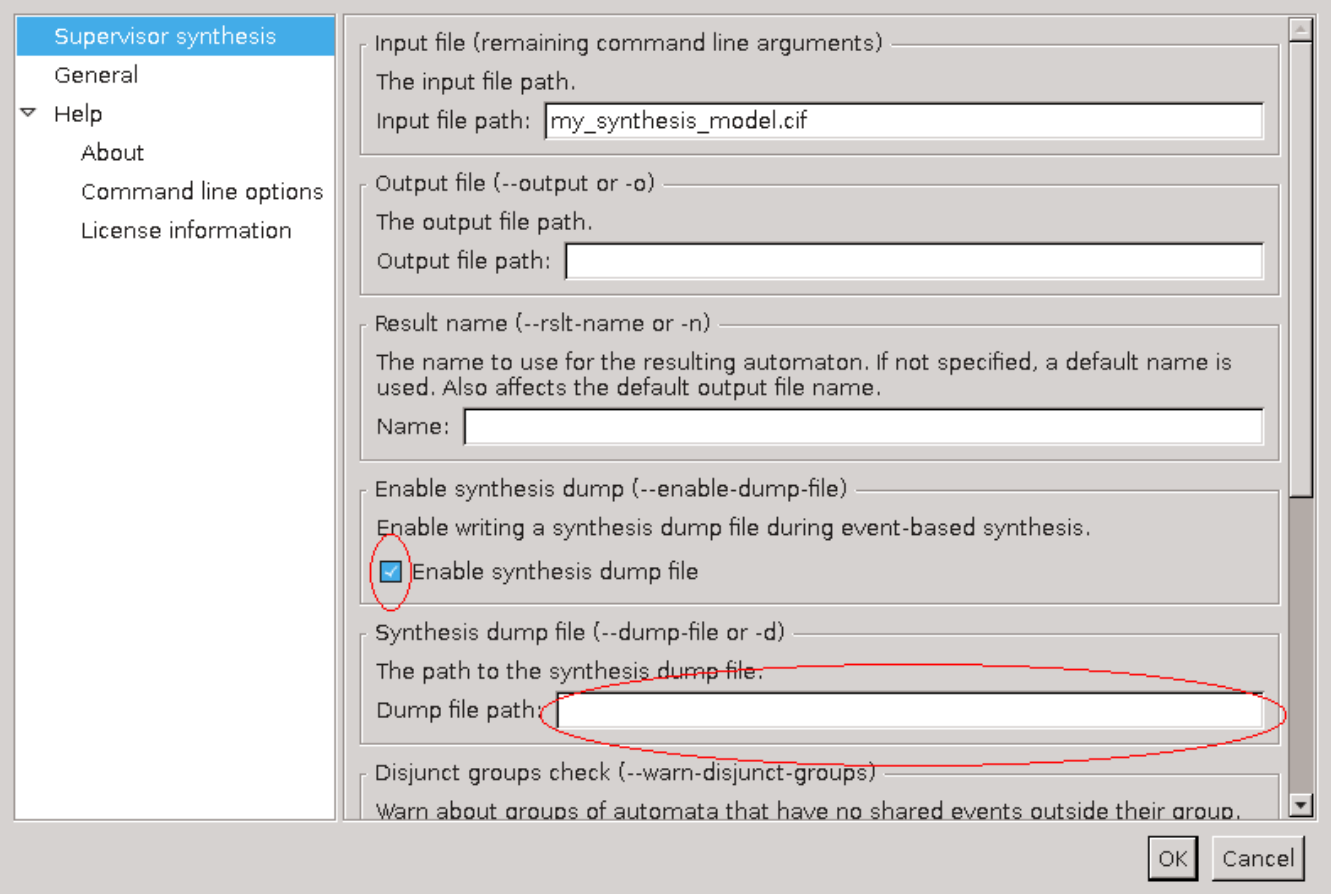
extension is removed (if present), and a `.synth_dump` file extension is added.

- *Disjunct groups check*: If enabled, the tool will report about groups of automata that share events in the group only.
- *Empty alphabet check*: If enabled, the tool will report about automata without events in their alphabet.
- *Marked deadlock location check*: If enabled, the tool will report about marked locations without outgoing edges.
- *Single use controllable check*: If enabled, the tool will report about controllable events that are used in exactly one automaton.

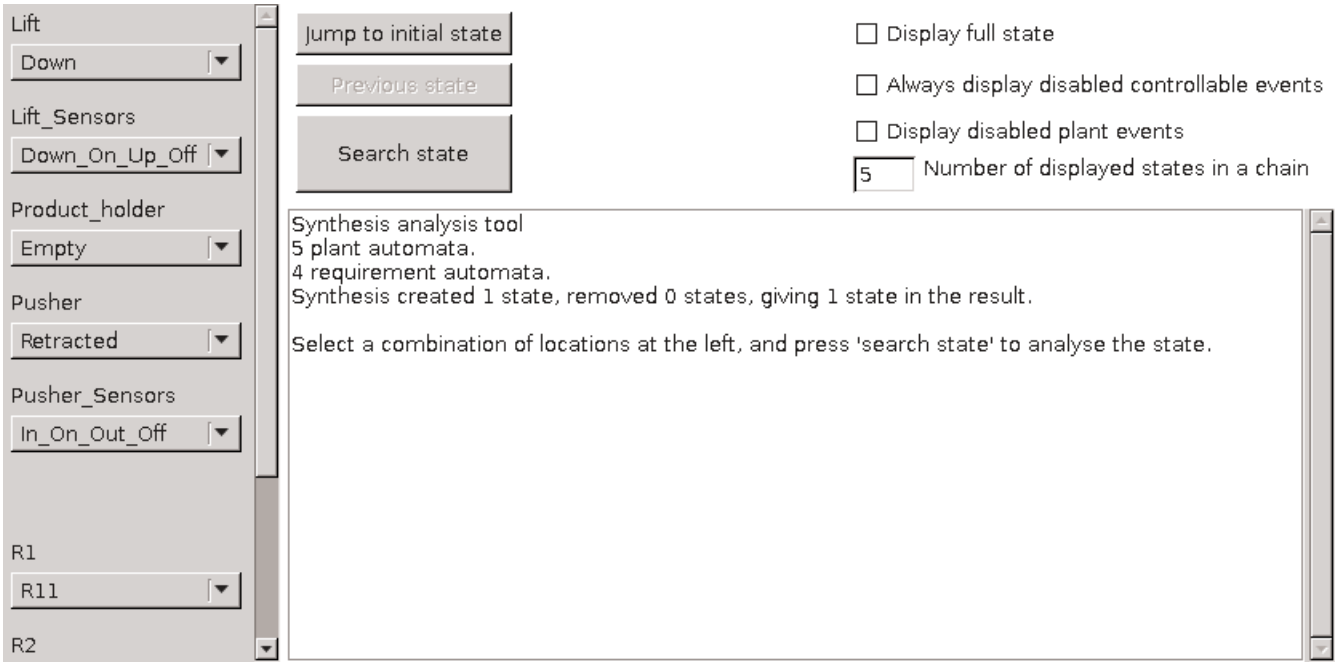
Event-based synthesis analysis

While developing a controller with supervisory controller synthesis, the result is not always what you expect. To assist, the synthesis analysis tool has been developed. Its primary purpose is to explain why certain states or edges have been removed.

To use the tool, enable creation of the *Synthesis Dump* file during synthesis, using the *Enable Synthesis dump* option or the *Synthesis dump file* option of the Supervisor synthesis tool.



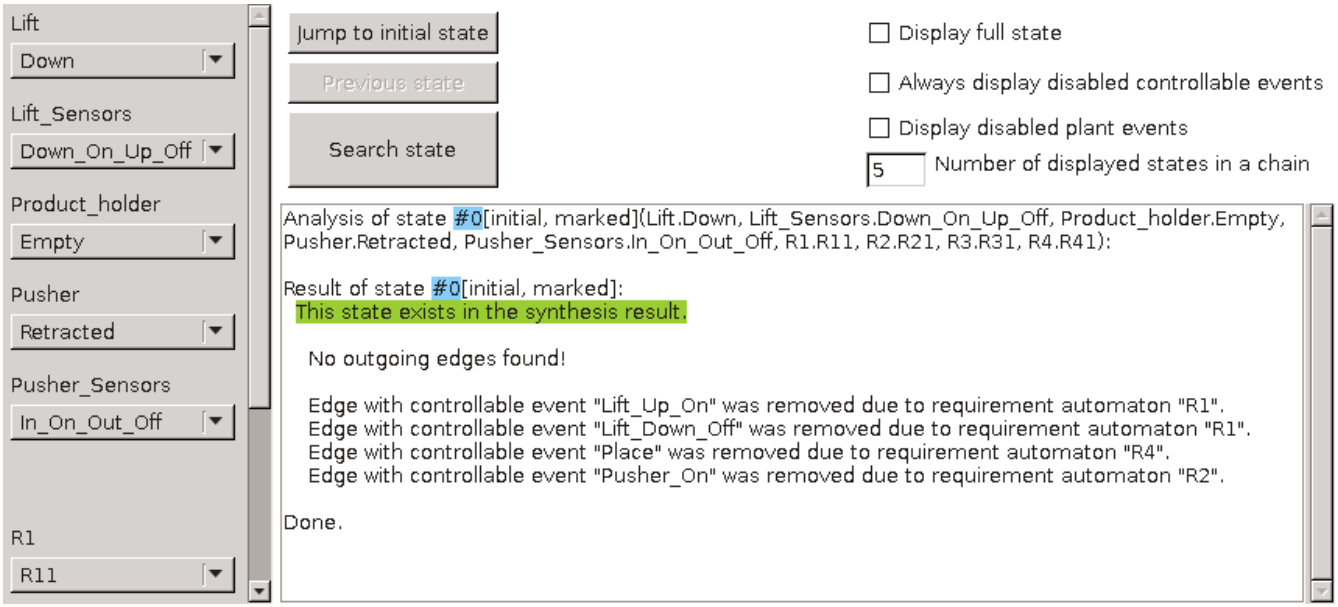
By double clicking the generated dump file, or by right-clicking the dump file and selecting **Analyze Synthesis Result**, the analysis tool is started. It loads the file (which can take a some time if the file is big), eventually giving the following display:



At the left, all automata with their initial location are shown. You can select a different supervisor state by changing these locations. At the top, in the middle, there are three buttons. The bottom **[Search state]** button is used to search the information about a supervisor state selected at the left. The **[Jump to initial state]** button resets the automata to their initial state, useful when you want to start again. The middle **[Previous state]** button is disabled now, but allows you to go back to the previous state.

At the right, a number of options are available in case the displayed information is not sufficient. These are explained later.

Below the buttons and options is the report area, which just displays some statistics. When you press **[Search state]**, the tool analyzes the selected state at the left. Depending on what happened with that state, different things may be shown. In this example, you get:

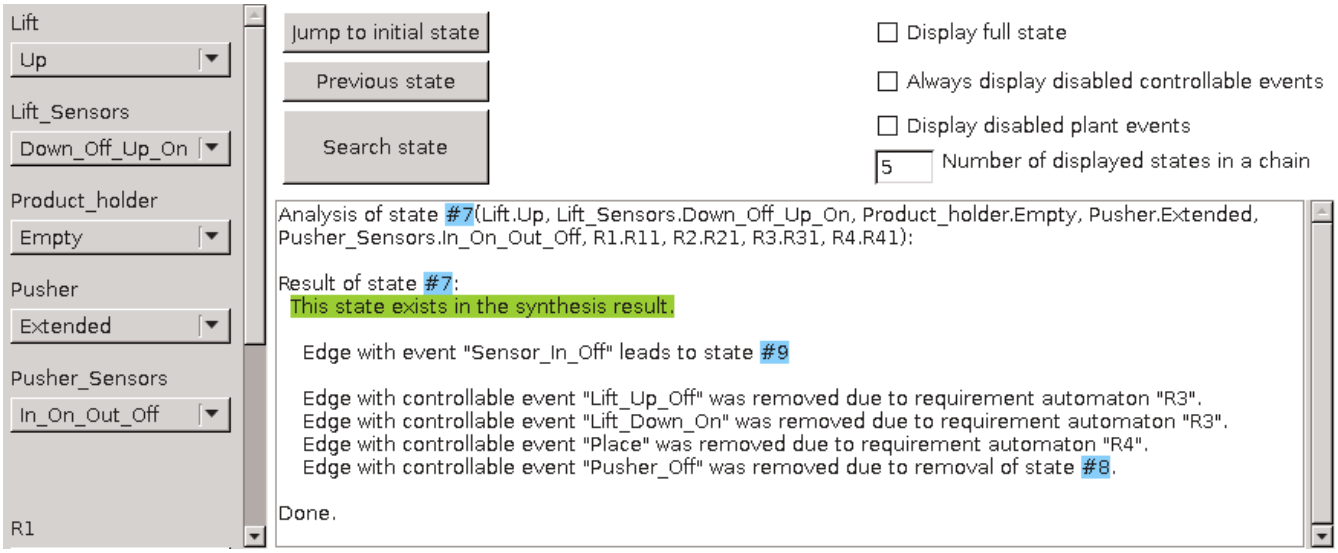


The report area changes to display the information about the selected state. It starts with the internal number of this state (`#0`), and its properties. The `initial` means this state is the initial state, `marked` means this state is marked. Then follows an explicit list of locations of every automaton. It is

the same list as the selected locations at the left, but more compact. Then the analysis result is shown. This state exists in the supervisor, and has no outgoing edges. In other words, this state is the only state in the result. Below it, the edges removed by the requirements are listed, together with the name of the requirement automaton that disabled it.

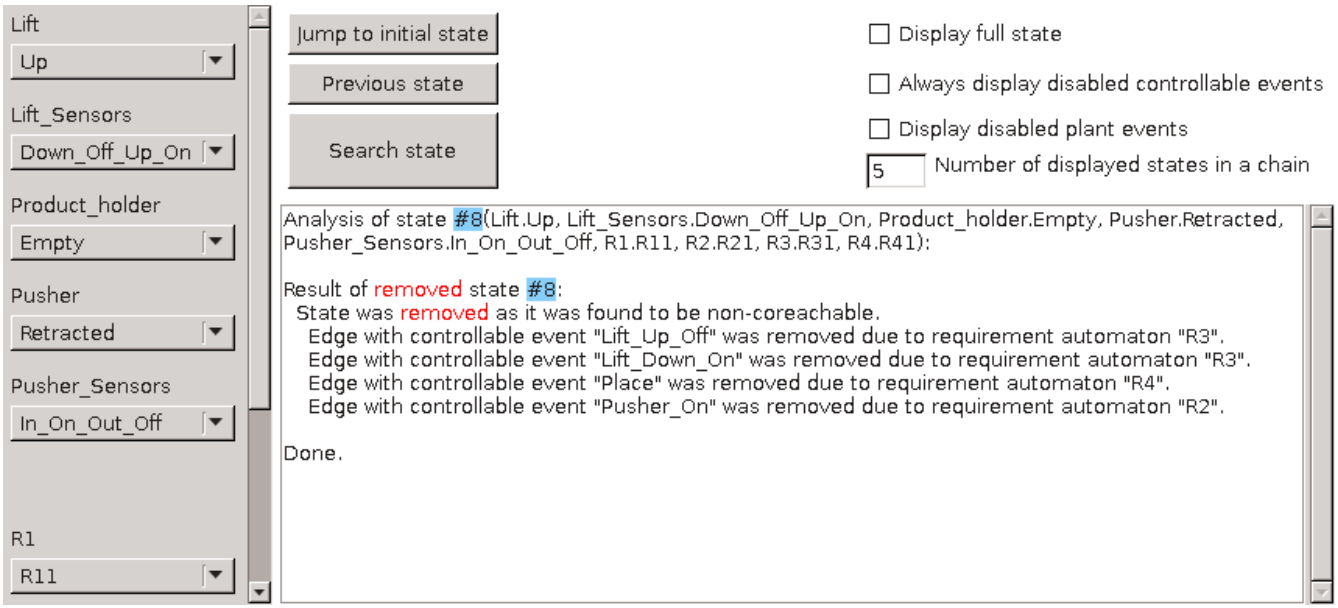
Not displayed here is the list of edges removed due to plant automata. These can be displayed by enabling the *Display disabled plant events* option at the top-right.

If a state does have outgoing edges, these are displayed as well:



State #7 has one outgoing edge, leading to state #9 using event `Sensor_In_Off`. The last line with the removed controllable events displays a removed edge because destination state #8 was removed.

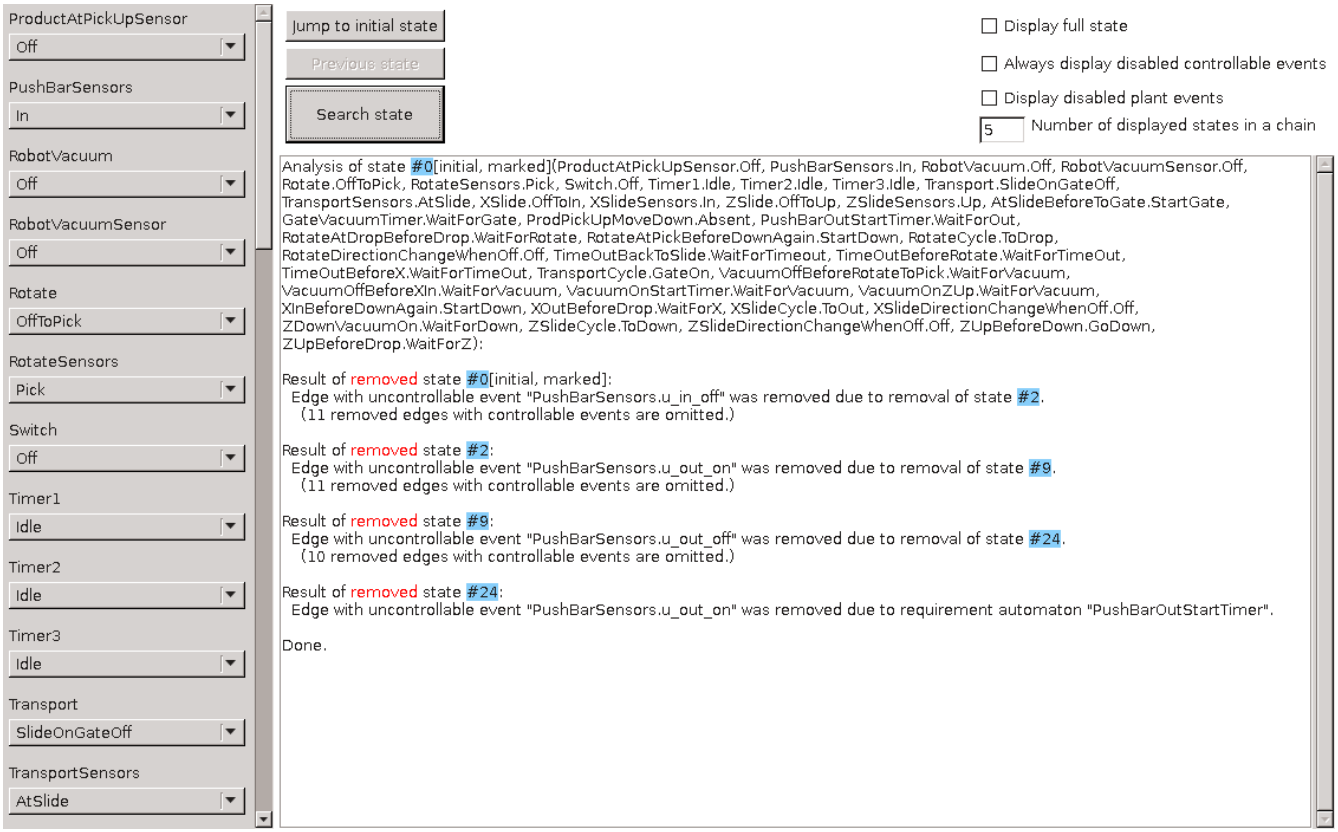
If you want to know what states #9 or #8 are, you can enable the *Display full state* option at the top-right, and the tool will display the full list of automaton locations with each state, just like the first line in the report. You can also go to these states, by double-clicking at the blue area. If you click at removed state #8 you get a display like:



As expected, the analysis tools displays that the state was removed, apparently because it was not co-reachable. Again you get the list of removed edges from this state, as an aid to decide which

requirement should be checked, if this was not expected.

The final screenshot shows what is displayed if a state is removed because another state is removed:



This is an example from a bigger system, where the supervisor is empty. The analysis of the initial state reveals that it was removed due to a removal of a chain of other states. State #24 was removed due to a requirement (reading from bottom to top), which caused the chain of states #9, #2, and eventually #0 to be removed.

Removed edges with controllable events are not displayed here, since these have no influence to the disappearing of the states. If you want, you can display them by toggling the *Always display disabled controllable events* option at the top-right.

In this example, the number of states in the chain to the disabling requirement was 4 states. This chain can be longer. Usually the middle part is not that interesting and it is omitted from the display, but if you want to see them, you can increase the number of displayed states at the top-right (which is by default 5). Another way to see them is to double-click at a state in the chain, and the analysis tool will display the chain starting from that state.

A situation that is not shown here, is that the analysis tool has no information about a state. It happens when the synthesis encountered a bad state on all paths to that state. In such a case, it displays that there is no information about the queried state.

Starting the synthesis analysis tool

The tool can be started in the following ways:

- In Eclipse, double-click a `.synth_dump` file generated by the supervisor synthesis dump.
- In Eclipse, right click a `.synth_dump` file in the *Project Explorer* tab or *Package Explorer* tab and choose **Analyze Synthesis Result**.
- Use the `cifsynthanalys` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifsynthanalys` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input Synthesis dump file.

Event-based nonconflicting check

The nonconflicting check verifies whether two or more automata are conflicting, that is, whether they together could result in non-coreachable states.

One practical application is to check a supervisor, either synthesized or manually created, against a plant. That is, when the supervisor is used to control the plant, will it always stay in the safe coreachable area? If not, at what point does it fail to do so?

Another practical application is to check several supervisors synthesized for parts of a system. That is, when the supervisors are used together to control a system, could that result in non-coreachable states? If so, which states are conflicting?

The tool takes a CIF file containing two or more automata, that must be both deterministic and trim. In addition, the general event-based restrictions apply as well.

The nonconflicting check tool produces a text file, listing the traces that lead to a conflicting state. If no such traces are found, the automata are nonconflicting.

Starting the nonconflicting check tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply nonconflicting check...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply nonconflicting check...**.
- Use the `cifncchk` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifncchk` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Report file*: The absolute or relative local file system path to the output report file with conflicts. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_conflicts.txt` file extension is added.

Event-based controllability check

The controllability check verifies whether a supervisor automaton does not disable edges with uncontrollable events of the (combined) plant automata. If the check fails, the tool reports where it fails. If the check succeeds, it reports the edges with controllable events that are disabled by the supervisor.

The tool takes a CIF file containing a `supervisor` automaton, and one or more `plant` automata. Besides the general event-based restrictions, the tool does not support:

- Having more than one `supervisor` automaton.
- Having no `plant` or `supervisor` automaton.
- Having an automaton with a different kind than `plant` or `supervisor`.
- Having a non-deterministic automaton.
- Events that are not controllable or uncontrollable.

The controllability check tool produces a report text file with its findings. It states whether the controllability property holds (no edges with uncontrollable events of the plant are disabled) or fails (one or more edges with uncontrollable events in the plant are disabled by the supervisor).

If the controllability property holds, the tool lists the disabled controllable events, which can be useful in the design process. If the property fails, the tool lists the edges that are disabled by the supervisor.

Starting the controllability check tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply controllability check...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply controllability check...**.
- Use the `cifctrlchk` tool in a ToolDef script. See the scripting documentation and tools overview page for details.

- Use the `cifctrlchk` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Report file*: The absolute or relative local file system path to the output report file with disabled events. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_disableds.txt` file extension is added.

Event-based language equivalence check

The language equivalence check verifies whether two automata produce the same language, that is, at every point they are marked in the same way and can produce the same events. Note that even automata with a different number of locations or edges can be language equivalent.

All reachable locations of the automata are checked. In particular, it also checks locations that are outside the marked behavior (that is, reachable locations that are not marked and are not on a path to a marked location). To avoid false positives, you may want to ensure that both automata are trim.

One practical application is to verify whether a manually created supervisor is equivalent to an automatically generated one. If they are not, a counter example is produced.

The tool takes a CIF file containing exactly two automata, that must be deterministic, have the same alphabet, and have an initial location. In addition, the general event-based restrictions apply as well.

The language equivalence check tool can produce three different results:

- The output stating `Automata have the same language` when the two automata are language equivalent.
- The output stating `Automata have a different language` when an event can be performed in a location in one automaton, while from the equivalent location in the other automaton the same event cannot be performed.
- The output stating `Automata have a different language` when a location in one automaton is marked, while the equivalent location in the other automaton is not marked.

In the last two cases, the equivalent location is derived by walking from the initial location to the location of interest in both automata using the same sequence of events. When two automata are not language equivalent, a counter example is produced that shows the sequence of events to reach the equivalent locations.

Starting the language equivalence check tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply language equivalence check...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply language equivalence check...**.
- Use the `ciflngeqv` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `ciflngeqv` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.

Event-based NFA to DFA automaton conversion

The event-based NFA to DFA automaton conversion takes a non-deterministic automaton and produces a deterministic automaton with the same language. It thus determinizes the NFA to a DFA.

The tool takes a CIF file containing one automaton. In addition, the general event-based restrictions apply as well.

The output is a CIF file containing a deterministic automaton that is language equivalent with the input automaton. The resulting automaton has the same supervisory kind as the input automaton. The resulting DFA is not necessarily minimal, but can be minimized using the DFA minimizer.

Starting the NFA to DFA automaton conversion tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply NFA to DFA automaton conversion...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply NFA to DFA automaton conversion...**.
- Use the `cifnfadfa` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifnfadfa` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_dfa.cif` file extension is added. The `dfa` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the projected automaton. If not specified, defaults to `dfa`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default. If the output CIF file has an automaton with only a single non-initial location, then the location does not get a state annotation, regardless of whether the option is enabled or not.

Event-based DFA minimization

Minimizing the number of locations of an automaton while preserving the language gives a canonical representation of the language, making it easier to compare automata. The DFA minimization tool computes such a minimal automaton.

The tool takes a CIF file containing one deterministic automaton with an initial location. In addition, the general event-based restrictions apply as well.

The DFA minimization tool produces an automaton with the same language, but with the minimal number of locations. If the original automaton already had the minimal number of locations, the result is the same as the input.

To minimize an NFA, first determinize it to a DFA.

Starting the DFA minimization tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply DFA minimization...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply DFA minimization...**.
- Use the `cifdfamin` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifdfamin` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_minimal.cif` file extension is added. The `minimal` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the minimized automaton. If not specified, defaults to `minimal`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default.

Event-based automaton projection

The event-based automaton projection takes an automaton and a subset of the events of its alphabet that should be preserved. The other events of the alphabet are considered to be internal steps and get replaced by `epsilon` at the edges. The algorithm produces a deterministic automaton with the subset as its alphabet that has the same language.

The tool takes a CIF file containing one automaton, and the names of the events that should be preserved. In addition, the general event-based restrictions apply as well.

The output is a CIF file containing a deterministic automaton that is language equivalent with the input automaton with respect to the set of preserved events. The resulting automaton has the same kind as the input automaton. The resulting automaton is a DFA that it is not necessarily minimal, but can be minimized using the DFA minimizer.

Starting the automaton projection tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply automaton projection...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply automaton projection...**.
- Use the `cifproj` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifproj` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Preserved events*: Comma and/or whitespace separated absolute names of events that should be preserved.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_projected.cif` file extension is added. The `projected` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the projected automaton. If not specified, defaults to `projected`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default. If the output CIF file has an automaton with only a single non-initial location, then the location does not get a state annotation, regardless of whether the option is enabled or not.

Event-based observer check

The event-based observer check takes an automaton, and a subset of the events of its alphabet that are observable. The check verifies whether the automaton after projection (see Event-based automaton projection) can derive the (abstracted) state of another component by synchronizing only on the observable events.

The tool takes a CIF file containing one automaton, and the names of the events that are observable. In addition, the general event-based restrictions apply as well.

The output is a report file that indicates whether the observer check property holds. If it fails, the events causing failure are also listed.

Starting the observer check tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply observer check...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply observer check...**.
- Use the `cifobschk` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifobschk` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Observable events*: Comma and/or whitespace separated absolute names of events that are observable.
- *Report file*: The absolute or relative local file system path to the output report file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_observation.txt` file extension is added.

Event-based automaton abstraction

The event-based automaton abstraction takes an automaton, and a subset of the events of its alphabet that are observable. The abstraction produces a non-deterministic abstracted automaton with the observable events as its alphabet, that is weakly bisimilar to the original automaton.

The tool takes a CIF file containing a single automaton, and the names of the events that are observable. Besides the general event-based restrictions, the tool does not support:

- Having no automata at all, or more than one automaton.
- Automata with only marked locations, or with only non-marked locations.
- Observable events that are not in the alphabet of the automaton.

The output is a CIF file with the abstracted automaton. The resulting automaton has the same kind as the input automaton.

Starting the automaton abstraction tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply automaton abstraction...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply automaton abstraction...**.
- Use the `cifabstr` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifabstr` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Observable events*: Comma and/or whitespace separated absolute names of events that are observable.
- *Output file*: The absolute or relative local file system path to the output CIF file. If not specified,

defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_abstracted.cif` file extension is added. The `abstracted` part of the default extension depends on the *Result name* option.

- *Result name*: The name to use for the abstracted automaton. If not specified, defaults to `abstracted`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default.

Event-based trim check

This check verifies that the automata in the given file are trim, that is, in each automaton, all the locations must be both reachable and coreachable.

The general event-based restrictions apply.

If the check fails for one or more automata, the tool reports for each automaton, whether all its locations are reachable and coreachable. It also lists the locations where the property fails.

Starting the trim check tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply trim check...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply trim check...**.
- Use the `ciftrimchk` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `ciftrimchk` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Report file*: The absolute or relative local file system path to the output report file with disabled events. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_trimcheck.txt` file extension is added.

Event-based trim

The trim tool makes an automaton trim by removing all locations that are not reachable or coreachable.

The input is a CIF file with one automaton, and the tool produces a new CIF file with the trim automaton. In addition, the general event-based restrictions apply as well.

If the automaton in the input was already trim, all locations are preserved.

Starting the trim tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Event-based synthesis tools › Apply trim...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Event-based synthesis tools › Apply trim...**.
- Use the `ciftrim` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `ciftrim` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_trim.cif` file extension is added. The `trim` part of the default extension depends on the *Result name* option.
- *Result name*: The name to use for the trim automaton. If not specified, defaults to `trim`. Also affects the *Output file* option.
- *Add state annotations*: Add state annotations to the locations of the automaton in the output CIF file. This option is enabled by default. If the output CIF file has an automaton with only a single non-initial location, then the location does not get a state annotation, regardless of whether the option is enabled or not.

4.3.4. CIF to Supremica transformer

The CIF to Supremica transformer can be used to transform CIF specifications to Supremica modules (`*.wmod` files). Supremica is a tool for synthesis of discrete event supervisors.

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF synthesis tools › Convert CIF to Supremica...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF synthesis tools › Convert CIF to Supremica...**.
- Use the `cif2supremica` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2supremica` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output Supremica module file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.wmod` file extension is added.
- *Supremica module name*: The name of the Supremica module. If not specified, defaults to the name of the output file, after removal of the `.wmod` extension (if present).
- *Eliminate enumerations*: Enable this option to eliminate enumerations and replace them with integers before transforming to Supremica. Disable this option to keep enumerations in the generated Supremica module. By default this option is disabled.

  Even though Supremica supports enumerations in its language, it doesn't support them in all its algorithms. Even if the original CIF transformation being transformed does not use enumerations, enumerations may still be generated internally during preprocessing.

Supported specifications

The CIF to Supremica transformer supports a subset of CIF specifications. The following restrictions apply:

- Kindless/regular automata (without a supervisory kind) are not supported. Only `plant`, `requirement`, and `supervisor` automata are supported.
- Events not declared as controllable or uncontrollable are not supported. This includes the `tau` event, both explicitly used on edges, as well as implicitly for edges without explicitly mentioned events.
- Initialization predicates outside of locations are not supported.
- Marker predicates outside of locations are only supported if they have the form `discrete_variable = marked_value`.
- Discrete variables with multiple marker predicates are not supported.
- Locations with initialization or marker predicates that are not trivially true or false are not

supported.

- Automata that do not have exactly one initial location are not supported.
- State invariants in locations are not supported.
- State invariants are only supported if they are requirement invariants.
- Kindless/regular state/event exclusion invariants (without a supervisory kind) are not supported. Only `plant`, `requirement`, and `supervisor` state/event exclusion invariants are supported.
- Discrete variables that have multiple potential initial values (`in {...}` or `in any`) are not supported.
- Continuous variables are not supported.
- Input variables are currently unsupported.
- Multi-assignments on edges (such as `do (x, y) := (1, 2)`) are not supported. However, it is allowed to use multiple assignments on an edge (such as `do x := 1, y := 2`).
- Partial variable assignments (such as `do x[0] := 5`) are not supported.
- Conditional updates on edges (such as `do if b: x := 5 end`) are not supported.
- Urgent locations and urgent edges are not supported.
- User-defined functions are not supported.
- Channels (events with data types) are not supported.
- Only the following data types are supported: boolean types, ranged integer types, and enumeration types.
- Only the following expressions are supported: boolean literal values, integer literal values, binary expressions (partially, see below), unary expressions (partially, see below), and references to constants, discrete variables, enumeration literals, and casts that don't change the type.
- Only the following binary operators are supported: logical equivalence (`<=>`), logical implication (`=>`), conjunction (`and`) on boolean operands, disjunction (`or`) on boolean operands, addition (`+`) on ranged integer operands, subtraction (`-`) on ranged integer operands, multiplication (`*`) on ranged integer operands, integer division (`div`) on ranged integer operands, integer modulus (`mod`) on ranged integer operands, equality (`=`), inequality (`!=`), less than (`<`) on ranged integer operands, less than or equal to (`<=`) on ranged integer operands, greater than (`>`) on ranged integer operands, and greater than or equal to (`>=`) on ranged integer operands.
- Only the following unary operators are supported: logical inverse (`not`), negation (`-`) on a ranged integer operand, and plus (`+`) on a ranged integer operand.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
- Annotations are ignored.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to

increase the subset of CIF specifications that can be transformed:

- Remove I/O declarations
- Remove position information
- Remove annotations
- Eliminate component definition/instantiation
- Eliminate state/event exclusion invariants
- Add default initial values
- Eliminate automaton `self` references
- Eliminate algebraic variables
- Eliminate the use of locations in expressions
- Eliminate type declarations
- Eliminate monitors
- Convert enumerations to integers (if enabled via the corresponding option)
- Simplify values
- Simplify others

Transformation result

The CIF automata kinds `plant`, `requirement`, and `supervisor` are mapped to Supremica kinds `Plant`, `Specification`, and `Supervisor`, respectively.

All generated Supremica automata are tagged as non-deterministic, as CIF automata may be non-deterministic.

Nameless CIF locations are given the name `X` in Supremica automata.

Supremica does not support boolean values. Instead of `true`, value `1` is used, and instead of `false`, value `0` is used. For boolean types (`bool`), ranged integer types (`int[0..1]`) are used.

The absolute names of all CIF objects (automata, variables, etc) are used, where the `.` characters are replaced by `:` characters. For enumeration literals, the non-absolute name of the enumeration literal is used, prefixed with `:lit:`.

No geometry is generated. When the generated Supremica file is opened in Supremica, Supremica will perform auto layouting.

Initialization

For the initial values of the variables, initialization predicates are generated (e.g. `x == 3`). Earlier versions of Supremica that support variables allowed deterministic initialization (only the initial

value), and non-deterministic initialization (initialization predicates). The most recent version only allows non-deterministic initialization, which is why we generate initialization predicates.

Marking

In CIF, if none of the locations of an automaton are indicated as marked, the automaton has no marked locations. In Supremica, if none of the locations of an automaton are indicated as marked (`:accepting`), all locations of the automaton are implicitly marked. When transforming a CIF automaton without any marked locations, a warning is printed to the console to inform the user of this difference.

In CIF, if none of the values of a variable is indicated as marked, the variable has no marked values. In Supremica, if none of the values of a variable is indicated as marked (`:accepting`), all values of the variable are implicitly marked. When transforming a CIF variable without any marked values, a warning is printed to the console to inform the user of this difference.

Location pointer variables that are automatically generated, such as the ones generated by the Eliminate the use of locations in expressions CIF to CIF transformation, have no marking and thus lead to warnings.

State invariants

If the CIF model being transformed contains state invariants, an uncontrollable `u_inv_bad` event is added to the Supremica module. It is renamed if the name is not unique. A plant automaton named `inv_plant` (renamed if not unique) is added as well. This plant enables the `u_inv_bad` event if and only if at least one of the invariants doesn't hold. A requirement automaton `inv_req` (renamed if not unique) is added as well. The requirement disables the event globally. Since blocking an uncontrollable plant event in a requirement is forbidden, synthesis will prevent such blockage (by disabling controllable events), thereby ensuring that the state invariants hold after synthesis.

Alphabets

In CIF, it is possible to explicitly specify the alphabet of an automaton. This alphabet may include more events than occur on the edges of the automaton, thereby globally disabling those additional events. Supremica automatically determines the alphabet automatically, based on the events that occur on the edges of the automaton, just like CIF does if no explicit alphabet is specified. If a CIF automaton with 'additional' events is transformed, a self loop is added to the initial state for each 'additional' event, with a `false` guard (`0` in Supremica). This ensures that the event occurs on an edge, and is thus part of the alphabet, but is not enabled.

Range semantics

In CIF, if an update of an edge results in out of range values of variables, the CIF specification is considered invalid, and the simulation will result in a runtime error. In Supremica, the simulation does not result in a runtime error, but instead the offending transition is disabled/forbidden.

Jumping semantics

Consider the following CIF model:

```
plant automaton p:
  controllable c_event;
  disc int[0..3] v = 0;

  location loc1:
    initial;
    marked;
    edge c_event do v := v + 1 goto loc2;

  location loc2:
    edge c_event               goto loc1;
end
```

There are two locations, and the edges allow for moving from one location to the other. Both edges use the same event. Every odd transition (the first transition, the third transition, etc), the value of variable `v` is increased by one. Every even transition, the value of variable `v` is not changed. After a few transitions, the simulation crashes due to overflow of variable `v`.

To ensure the same semantics in Supremica, such that for every even transition the value of variable `v` does not change, the CIF model is modified by the transformation, to the following:

```
plant automaton p:
  controllable c_event;
  disc int[0..3] v = 0;

  location loc1:
    initial;
    marked;
    edge c_event do v := v + 1 goto loc2;

  location loc2:
    edge c_event do v := v     goto loc1;
end
```

Users of the transformation don't have to do anything themselves, as the tool automatically adds the dummy assignments as needed. If the tool would not have added such dummy assignments, then the value of variable `v` would have been able to *jump* to any value (in its range `[0..3]`) for

every even transition, according to the Supremica semantics.

4.4. Simulation, validation, and verification tools

4.4.1. CIF simulator

The CIF simulator can be used to explore the state space of CIF specifications, either interactively, or using a more automated approach. Powerful visualization features allow for interactive visualization-based validation of the CIF specification.

Basics

- Supported specifications
- Starting the simulator
- Initialization of the CIF specification
- Terminating the simulator
- Simulation of traces
- Repeated simulation

Input/output

- Input modes
- Non-urgent events
- Simulation output

Advanced

- Simulator options
- Simulation performance
- ODE solver (integration and guard/event detection)
- Java compiler

Miscellaneous

- Execution mode
- Complete mode
- External user-defined functions
- Profiling
- Stochastic distributions and seeds

Developers

- Maximum time point tolerance
- Debugging generated code

- Test mode

Simulation basics

Supported specifications

The CIF simulator supports a subset of CIF specifications. The following restrictions apply:

- Time dependent state invariants are not supported. This applies only to state invariants. State/event exclusion invariants are supported.

Specifications with component definitions/instantiations are not natively supported by the CIF simulator. Therefore, they are automatically eliminated by the simulator, as a preprocessing step, using the CIF to CIF transformation to eliminate component definition/instantiation.

Automata with multiple possible initial locations, input variables, and discrete variables with multiple possible initial values (including `any`) are not supported, unless additional initialization is provided.

Input variables are in principle simulated as having a constant value. It's possible to change their values by having SVG input mappings update them, or by using the `input` commands in a trace file. Alternatively, input variables may first be merged with other variables that provide their values.

The controllability of events is ignored by the simulator, as are marker predicates.

All automata are simulated as plants. That is, all automaton kinds are ignored by the simulator. However, simulating requirements as plants may lead to unexpected results. Therefore, the simulator prints warnings to the console, whenever requirements are simulated. It is highly recommended to first apply supervisor synthesis to the specification, and simulate the resulting specification using the simulator. Alternatively, apply verification to the specification, remove the verified requirements, and simulate the resulting specification using the simulator.

Similar to requirement automata, the simulator warns about simulation of requirement invariants.

Annotations are ignored.

Starting the simulator

The CIF simulator can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF simulation, validation and verification tools › Simulate CIF specification...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF simulation, validation and verification tools › Simulate CIF specification...**.
- Use the `cifsim` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifsim` command line tool.

Once you start the simulator, an option dialog will appear, where you can configure the simulation options.

Instead of a CIF model (e.g. a `.cif` or `.cifx` file), simulation can also be started for a compiled code file (e.g. a `.cifcode` file). For compiled code files, an additional option is available. The simulator can also be started by double clicking such a file.

Initialization of the CIF specification

CIF supports discrete variables with multiple potential initial values, as well as discrete variables with an arbitrary initial value, and input variables without an explicit value. Furthermore, CIF supports automata with multiple potential initial locations. The simulator however, needs to know how to initialize the specification. The initial values of discrete variables and input variables, and the initial locations of automata, can be specified using the *CIF specification initialization* option (*Simulator* category). Initialization can be provided using the option, per discrete variable, input variable or automaton.

For discrete variables and input variables, initialization is of the form `var:value`, where `var` is the absolute name of a discrete variable or input variable, and `value` is a CIF literal that represents a valid initial value for the variable. Providing an initialization with a function type or a distribution type, is not supported.

For an automaton, initialization is of the form `aut:loc`, where `aut` is the absolute name of an automaton, and `loc` is the name of a location of that automaton. Providing an initialization for an automata with only one nameless location, is not supported.

It is not allowed to provide multiple initializations for a single variable or automaton.

If insufficient initializations are provided, variables and automata may not be initialized. That is, the simulator may not be able to choose between the multiple potential initial values or locations. The simulator will report such problems, indicating the specification is unsupported without additional or alternative initializations.

It is also possible that initializations in the CIF specification itself are incomplete or conflicting, the provided initializations are conflicting, or their combination is conflicting. Furthermore, initializations may be provided for values or locations that are not potential initial values or locations. In all such cases, initialization fails. The simulator will report such problems, providing detailed information about what caused the initialization failure.

Terminating the simulator

The CIF simulator can terminate for various reasons. This page explains each of them, as well as some other information related to termination of the simulator.

Initialization failure

If initialization of the specification fails, the simulation will terminate immediately.

Deadlock

If simulation results in deadlock, no more transitions are possible, and the simulation will terminate.

Simulation does not immediately terminate when deadlock occurs, if an interactive input mode is used and history is enabled. Instead, the user is asked whether to terminate the simulation, undo one or more transitions, or reset the simulation.

Simulation end time

Using the *Simulation end time* option (*Simulator* category), the end time of the simulation can be configured. By setting a simulation end time, you ensure that when the value of variable `time` reaches that value, the simulation terminates.

By default the simulation end time is set to 'infinite', which means that simulation termination is not limited by a simulation end time. Settings a simulation end time also limits the maximum duration of a time transition.

Simulation does not immediately terminate when the configured end time is reached, if an interactive input mode is used and history is enabled. Instead, the user is asked whether to terminate the simulation, undo one or more transitions, or reset the simulation.

Runtime error

If simulation results in a runtime error, the simulation will terminate immediately.

End of trace file

If the trace input mode is used, and no more commands are available in the trace file, the simulation will terminate immediately.

User-requested termination

The user can manually request termination of the simulation in several ways.

Termination requests have no effect while the simulator interactively asks for input from the console. However, once input is provided via the console and `Enter` is pressed, the termination request will be processed.

During the execution of external user-defined functions, termination requests are not processed. If an external user-defined function doesn't terminate, the simulator can't be terminated by the user.

Termination requests issued between intermediate frames are delayed until the next intermediate frame.

Interactive console input

Termination of the simulation can be requested, when using the interactive console input mode, by typing `q` and pressing `Enter`, whenever the simulator interactively asks for input via the console.

Interactive GUI input

Termination of the simulation can be requested, when using the interactive interactive GUI input mode, by closing the GUI of the interactive GUI input component, when the simulator asks for input via that GUI, or by clicking the console termination button.

Applications view

The *Applications* view has several termination features that may terminate any application, including the CIF simulator. See the documentation for that view, for details.

Console termination button

Simulation can be terminated at any time, by using the *Terminate* button (■), located at the upper right corner of the console.

Note however that if the console does not have the focus, this button may not be visible. If the button is not visible, click somewhere in the console to make the button appear. If even then the button is still not available, it may still appear if you *Maximize* the console. Due to these issues, use of the termination features of the *Applications* view is recommended over using the *Terminate* button of the console. See the documentation for that view, for details.

Closing last real-time visualizer

If the last remaining real-time output component (visualizer) is closed by the user, the simulator will be asked to terminate as soon as possible.

Confirming termination

When simulating within Eclipse, all visualizers remain open, after the simulation has terminated. This allows you to observe the final state of the simulation, before manually closing the visualizers.

When simulating on the command line (outside of Eclipse), termination of the simulator means that the visualizers are also closed. To allow you to observe the final state of the simulation, you are asked to confirm termination of the simulator, and therefore closing of the visualizers, by pressing `Enter` at the console.

By default, you are only asked to confirm termination of the simulator when simulating on the command line (outside of Eclipse), at least one visualizer is visible at the end of the simulation, simulation did not result in a runtime error, and test mode is disabled. The *Ask to confirm simulator termination* option (*Simulator* category) can be used to configure whether the simulator asks the user to confirm termination of the simulator, terminate automatically, or decides by itself whether to ask the user for confirmation or terminate automatically. The latter (letting the simulator decide by itself) is the default.

Simulation of traces

The CIF simulator simulates one particular trace through the state space of the specification. To better understand what the previous sentence means, this page further explains each of those concepts.

State

Automata consist of one or more locations. However, at any time, an automaton can only be in *one* of its locations. This location is called the *current* or *active* location. The current location of an automaton is also called the *state* of the automaton.

Similar to having multiple locations in automata, variables usually have many possible values. However, similar to automata having only one current location, variables can not have two values at the same time.

The current location of each of the automata (i.e. the states of the automata), together with the current values of the variables (all discrete, input and continuous variables, including variable `time`), is called the *state* of the specification. The state of the specification (or simply 'the state'), is all the information that needs to be maintained about the history of the simulation, going forward.

For instance, consider the following specification:

```
automaton button:
  event pushed, released;

  location Released:
    initial;
    edge pushed goto Pushed;

  location Pushed:
    edge released goto Released;
end

automaton machine:
  event producing, produced;

  location Idle:
    initial;
    edge producing when button.Pushed goto Producing;

  location Producing:
    edge produced goto Idle;
end
```

We have two automata, `button` and `machine`. Pushing the button (event `button.pushed`) turns on the machine. Releasing the button (event `button.released`) turns it off. Initially, the machine is `Idle`. While the button is pushed (location `button.Pushed` is the current location of automaton `button`), the machine can start to create a product (event `machine.producing`). Once the machine has produced a product (event `machine.produced`), the machine becomes idle again. Note that once the machine starts producing a product, it always finishes producing it, even if it is turned off in the mean time.

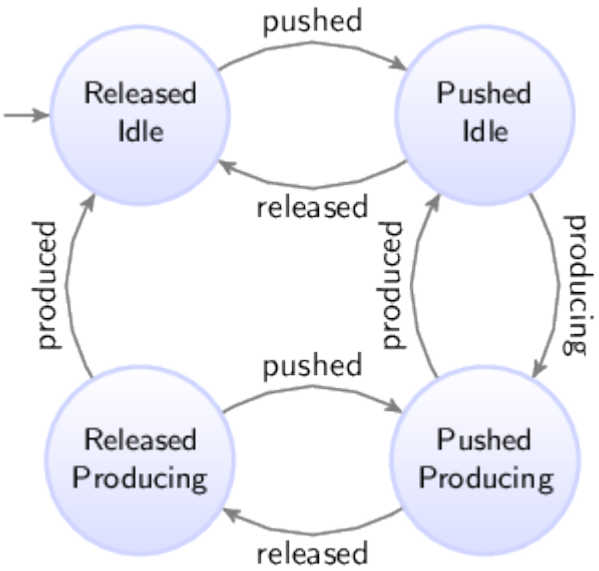
The state of automaton `button` is either of its locations: `button.Released` or `button.Pushed`. Similarly,

the state of automaton `machine` is `machine.Idle` or `machine.Producing`. Initially, the system is in state `button: Released, machine: Idle, time: 0.0`. For the remainder of variable `time` will omitted from the state, for brevity only. Furthermore, also for brevity only, we'll omit the names of the automata. Therefore, the initial state is `Released, Idle`.

State space

The *state space* of a specification consists all the possible states of the specification, connected by the transitions via which they can be reached.

For the example above, the state space is (event names are abbreviated):



Since both automata can be in two states, the state space in this case consists of four states. From the initial state (the upper left state), the button can be pushed, leading to the upper right state, where the button is pushed, while the machine has not yet started to produce a product. The machine can then start to produce a product (going to the lower right state). If however the button is released before the machine can respond by starting to produce a product, we go back to the upper left state. When the machine finishes creation of the product, we go from the lower right state back to the upper right state. If on the other hand the button is released, while the product is still being produced, we go from the lower right state to the lower left state. If we then push the button again, we go back to the lower right state. If instead the button is not pushed, and the machine finishes producing the product, we go from the lower left state back to the upper left state.

Traces

A *trace* is a sequence of transitions, from the initial state, through the state space.

In state spaces, most states usually have multiple outgoing transitions. This means there is *choice*. That is, it is possible to choose to which next state to go. Furthermore, it is usually possible to keep taking transitions forever, which means we can have infinite traces.

For the example above, some of the possible traces are shown below. Only the first five transitions

of the traces are shown. State names are abbreviated, to the first letters of the names of the locations.

- RI → `pushed` → PI → `released` → RI → `pushed` → PI → `released` → RI → `pushed` → PI → ...
- RI → `pushed` → PI → `producing` → PP → `produced` → PI → `producing` → PP → `produced` → PI → ...
- RI → `pushed` → PI → `producing` → PP → `released` → RP → `produced` → RI → `pushed` → PI → ...

In the first trace, we push the button, release it, push it again, release it, and push it again, all without ever starting to produce a product. This traces shows us what happens if the button is constantly being pushed and released, as quickly as possible, without the machine being able to respond to this, by starting to produce a product.

In the second trace, we push the button, we start to produce a product, we finish producing it, start to produce another product (the button is still pushed), finish producing the second product, and start to produce a third product. This is a trace of a typical usage scenario, where we start the machine, and the machine keeps running.

In the third trace, we push the button, start to produce a product, release the button before the product is finished, finish producing the product, and push the button again. This trace is another typical usage scenario, which also includes turning off the machine.

Simulation

As stated at the top of the page: *'The CIF simulator simulates one particular trace through the state space of the specification'*. To see why this is the case, we take a look at the main simulation loop, as used by the simulator:

- Calculate the initial state, and set it as the current state.
- Forever, do:
  - If the user-provided simulation end time is reached, stop simulation.
  - Calculate the possible transitions for the current state.
  - If no transitions are possible (deadlock), stop simulation.
  - Choose one of the possible transitions.
  - Take the chosen transition, and set its target state as the new current state.

While this main simulation loop is simplified with respect to the real implementation, it gives some insight into the inner workings of the simulator. The simulator keeps taking transitions. Once a transition is taken, the current state is updated to the target state of the transition. This means that the other possible transitions (the ones that were not chosen), are not taken. Therefore, if we want to take a different transition, we should restart simulation from the initial state, and make different choices. That is, if we want to simulate a different trace, we perform another simulation.

Validation

The simulator can be used to gain confidence in the correctness of the specification. By simulating various traces, we can observe what happens in different scenarios (use cases). Since the number of traces if often infinite, covering the entire state space, and all possible traces, is impossible. However, by wisely choosing the traces we simulate, we can cover a large part of the state space.

It should be clear by now, that simulating a single trace is almost never enough to conclude that your specification is 'correct'. Different traces lead to different behavior, and only by testing enough traces, and thus covering enough of the system's behavior, can you conclude that your specification works as expected (for those traces).

Repeated simulation

When simulating a CIF model, the simulator reads the CIF file, checks it for validity, generates Java code, compiles the code, and uses the compiled code using the actual simulation. All of this takes a bit of time, or even quite some time, depending on the size of the model. This situation is not ideal for repeated simulations, for instance when trying different scenarios or use cases, when simulating with different options, or when repeating simulations with various different initializations or due to stochastic behavior. The startup time for the first simulation is also present for the second simulation, etc.

It would be much better to generate and compile the code just once, and then simulate multiple times using the already compiled code, significantly reducing the startup time of the simulations. This is especially useful for automated repeated simulations, where the same simulation model, either with different options (for instance different initializations) or with the same options for a model with stochastics, is simulated multiple times and the results of simulation are for instance compared or aggregated in some way.

Compiling just once

The simulator can be used to generate code and compile it only once. By enabling the *Compile only* option (*Compiler* category), the simulator will perform all steps needed to generate and compile the code, and will then save the compiled code to a file. After saving the file, the simulator will terminate, not simulating the model.

By default, the input file name (e.g. `some_file.cif`) is used to construct the name of the simulator compiled Java code file, by removing the `.cif` file extension if present, and adding a `.cifcode` file extension to it (e.g. resulting in `some_file.cifcode`). It is possible to change this file name by using the *Compiled code file* option (*Compiler* category). If used, it specifies the absolute or relative local file system path to the compiled Java code file.

The compiled code file can be put in any directory. It does not have to be in the same directory as the original CIF model. However, make sure that any additional files, such as SVG images for visualization with the SVG visualizer, remain in their original locations. They will be resolved relative to the original location of the CIF model that was compiled.

Simulating using a compiled code file

There are several ways to start simulation with a compiled code file. In fact, all the ways to start simulation for normal CIF models (e.g. `.cif` files), apply to compiled code files (e.g. `.cifcode` file) as well. The simulator checks whether the input file name ends with `.cifcode` and then loads the compiled code file, instead of treating the input model as CIF model.

Alternatively, a `.cifcode` file can be double clicked in the Eclipse *Project Explorer* tab or *Package Explorer* tab, to immediately start simulation. An option dialog is shown to configure any simulator options.

A warning is printed to the console if the compiled code file was generated by a different version of the simulator than the one used to simulate it. Ignoring the warning may lead to unexpected simulation results, or even simulator crashes.

Option effects and priorities

Most simulator options have no effect when the *Compile only* option (*Compiler* category) is enabled, as after code is generated, compiled, and saved to a file, the simulator terminates, and no actual simulation is performed.

The *Compile only* option (*Compiler* category) has no effect when debugging generated code, by enabling the *Load debug code* option (*Simulator* category).

Example

As an example, consider the following CIF model in file `machine.cif`:

```
event enter;
event processed;
event exit;

automaton machine:
  disc int count = 0;
  disc real process_time;
  disc dist real d = normal(10, 7);
  cont t der 1.0;

  location waiting:
    initial;
    edge enter do (process_time, d) := sample d, t := 0 goto processing;

  location processing:
    edge processed when t >= process_time goto done;

  location done:
    edge exit do count := count + 1 goto waiting;
end

print machine.count for final;
```

The model consists of a machine that processes products. Initially, the machine is waiting. Once a product enters the system, the machine will start processing. Once the product is processed, the machine is done and will be waiting for the next product to enter. The time it takes to process a product varies with each product. A normal distribution with a mean of 10 and a variance of 7 is used to model the varying processing times.

The print declaration at the end of the model is used to print the number of processed products at the end of the simulation.

Consider the following ToolDef script:

```
from "lib:cif" import *;

cifsim("machine.cif --compile-only=on");

rmfile("out.txt", force=true);

for x in range(1000):
    string args = "machine.cifcode -i auto -a first -t 100 -o print";
    cifsim(args, stdout="out.txt", appendOut=true);
end

double minimum = 9999;
double maximum = -9999;
for line in readlines("out.txt"):
    double value = <double>line;
    minimum = min(minimum, value);
    maximum = max(maximum, value);
end

outln("Minimum: %.2f", minimum);
outln("Maximum: %.2f", maximum);
```

The import statement makes the CIF tools available. The first cifsim command generates code for the machine.cif model, compiles it, and saves it to machine.cifcode. The rmfile command removes the out.txt file, in case it was left over from a previous run of the script.

The body of the for statement is executed 1000 times. Each time, the cifsim command is used to simulate the model, using the already compiled code in machine.cifcode. The simulation uses automatic mode due to -i auto, always chooses the first available transition due to -a first, ends at time 100 due to -t 100, and only outputs text for print declarations due to -o print. The output of the simulation is not written to the console, but to out.txt due to the use of the stdout parameter of the cifsim tool. The appendOut parameter is enabled to ensure that the output of each simulation is appended (added) to the out.txt file instead of overwriting the file for each simulation. After the simulations, the out.txt file will contain the number of processed products for each of the simulation experiments.

The remainder of the script computes the minimum and maximum number of products (minimum and maximum throughput) of the machine, as observed from the experiments. Both the minimum

and `maximum` are initialized. Then each line of the `out.txt` file is processed. The numeric value is obtained by casting the text of the line to a `double` number. Then the `minimum` and `maximum` are updated as needed. After the minimum and maximum are calculated, they are printed to the console as floating point numbers with two digits after the decimal mark.

The output of the execution of the script could look like this, but may vary given that stochastics are used:

```
Minimum: 7.00
Maximum: 12.00
```

Applications view

The *Applications* view shows the current status of each running application. For repeated simulation, lots of applications may be started in a short period of time. Also, a lot of status updates need to be processed by the view in a short amount of time. This may lead to many updates of the GUI of the *Applications* view, which could potentially reduce the responsiveness of Eclipse, and decrease the performance of the simulation. If you experience such problems, simply close the *Applications* view while running a repeated simulation.

Simulation input/output

Input modes

The CIF simulator simulates traces through the state space of a specification. If multiple transitions are possible, a choice must be made. The simulator defers this choice to an *input component*.

The primary responsibility or an input component is to choose the transition to take, from the given possible transitions, calculated for the current state. If the chosen transition is a time transition, the input component must also decide the amount of time to delay.

The input component to use for the simulation, depends on the *input mode*, which can be configured via the *Input mode* option (*Input* category). The simulator has several input modes:

- Interactive console input mode
- Interactive GUI input mode
- Automatic input mode
- Trace input mode
- SVG input mode

Only one input mode, and thus one input component, can be used for each simulation. By default, the *GUI input mode* is used.

The following information on other topics, related to input modes, is available:

- Semi-automatic mode

- State history (reset, undo)

Interactive console input mode

In the interactive console input mode, the simulator interactively asks the user to choose the transitions to take, etc. The questions are asked via text printed to the console. The user enters his/her choice via the console as well.

Choosing a transition to take

The following is an example of the simulator interactively asking the user to choose a possible transition:

```
Possible transitions:
  #1: event machine1.c_start
  #2: event machine2.c_start
  #3: delay for (0 .. 100.0]

Select a transition ([1..3]). Enter q to quit. Enter h for help and additional
commands. Confirm with <ENTER>:
```

Here, three possible transitions are given, numbered `1` to `3`. For your convenience, they are sorted alphabetically on the names of the events, with event `tau` always at the end, just before time transitions.

The user should enter a command, and confirm that choice by pressing `Enter`. Several command are available:

- Entering a number in the given range, corresponding to the transition with that number, will result in that specific transition being taken. So, entering `2` and pressing `Enter`, will result in the simulator taking the transition for event `machine2.c_start`. The valid choices for the transitions are indicated by the range `[1..3]`. In case of an invalid number, an error message is printed to the console, and the user can choose again.
- Entering nothing, and only pressing `Enter`, defaults to choosing transition `1`.
- Entering `r` or `R`, and confirming that by pressing `Enter`, will reset the simulation to the initial state. If reset is not enabled, or not possible for the current state, an error message is printed and the user can choose again.
- Entering `u` or `U`, and confirming that by pressing `Enter`, will undo one transition. If undo is not enabled, or not possible for the current state, an error message is printed and the user can choose again.
- Entering `u <n>` or `U <n>`, with `<n>` a positive number, and confirming that by pressing `Enter`, will undo `n` transitions. If undo is not enabled, it is not possible to undo `n` transitions for the current state, an error message is printed and the user can choose again.
- Entering `q` or `Q`, and confirming that by pressing `Enter`, will terminate the simulation.
- Entering `h` or `H`, and confirming that by pressing `Enter`, will result in the simulator printing a short help text to the console, that shows the available commands. The user is then again asked

to choose.

If a choice is made that is invalid or is not available for the current state, an error message is printed to the console, and the user is asked to make an other choice.

Choosing a time transition duration

If the user chooses a time transition (choice `3` in the above example), the simulator subsequently asks for the duration of the time delay:

```
Select a duration from (0 .. 100.0]. Empty choice equals maximum delay. Enter q to
quit. Confirm with <ENTER>:
```

Here, the duration of the time delay is to be entered. In this case, the duration must be in the `(0 .. 100.0]` interval. That is, it must be larger than zero, and less or equal to `100.0`. Both integer and real values are allowed. As before, confirm with `Enter`.

Pressing `Enter` without having entered anything, defaults to the maximum delay, which in this case is `100.0`. Entering `q` or `Q`, and confirming that by pressing `Enter`, will terminate the simulation.

If a choice is made that is invalid, an error message is printed to the console, and the user is asked to make an other choice.

Choosing between termination or going back

If no transitions are possible, for instance due to deadlock or reaching the end-user provided simulation end time, the simulator may interactively ask the user to choose between termination of the simulation, resetting the simulation to the initial state, or undoing some transitions. This choice is only presented if either reset or undo is enabled, as otherwise there is no choice to make.

The following is an example of the simulator interactively asking the user to choose between termination and going back:

```
Simulation resulted in deadlock. Enter q to quit. Enter h for help and additional
commands. Confirm with <ENTER>:
```

The user should enter a command, and confirm that choice by pressing `Enter`. Several command are available:

- Entering `r` or `R`, and confirming that by pressing `Enter`, will reset the simulation to the initial state. If reset is not enabled, or not possible for the current state, an error message is printed and the user can choose again.
- Entering `u` or `U`, and confirming that by pressing `Enter`, will undo one transition. If undo is not enabled, or not possible for the current state, an error message is printed and the user can choose again.
- Entering `u <n>` or `U <n>`, with `<n>` a positive number, and confirming that by pressing `Enter`, will undo `n` transitions. If undo is not enabled, it is not possible to undo `n` transitions for the current

state, an error message is printed and the user can choose again.

- Entering `q` or `Q`, and confirming that by pressing `Enter`, will terminate the simulation.
- Entering `h` or `H`, and confirming that by pressing `Enter`, will result in the simulator printing a short help text to the console, that shows the available commands. The user is then again asked to choose.

If a choice is made that is invalid or is not available for the current state, an error message is printed to the console, and the user is asked to make an other choice.

Semi-automatic mode

By default, in the interactive console input mode, the user is asked to choose, even if for instance there is only one possible transition. Having to choose for every transition can become tiresome. To reduce the number of situations where a choice is required, the semi-automatic mode is available.

Interactive GUI input mode

In the interactive GUI input mode, the simulator interactively asks the user to choose the transitions to take, etc. Choices are presented using a GUI (Graphical User Interface). The user makes choices via the GUI as well.

Layout

The following is an example of the simulator interactively asking the user to choose a possible transition:



The events of the system are vertically listed, in alphabetical order, with event `tau` at the end, just before the **[time delay]**, **[reset]**, and **[undo]**. If there are too many elements to fit vertically, more columns are used. If things still don't fit, a horizontal scroll bar is shown.

Availability

Choices for events are only present if the events are interactively chosen. If they are (semi-)automatically chosen, they are not shown in the GUI. Similarly, choices for *time delay* are only shown if time transitions or their durations are interactively chosen. For more information, see the Semi-automatic mode section below.

For event `tau`, the choices are additionally only shown if there is at least one `tau` edge in the specification.

The choice to *reset* the simulation is only shown if resetting the simulation is enabled. Similarly, the choice to *undo* transitions is only shown if undo is enabled.

If none of the choices are to be shown in the GUI, the GUI itself is not shown at all.

Buttons

For each element, there are two buttons. The left button shows the name of the event, or a description of the choice (e.g. *time delay*, *reset*), and can be used to quickly choose a transition. The second button shows an arrow, and can be used to show a popup menu, presenting additional choices. What exactly is chosen when clicking the button, and what exactly is shown in the popup menu, differs for the different choices. The sections below provide further details.

If no choice needs to be made at the moment, all buttons are disabled. If a choice needs to be made, only those buttons that represent active choices are enabled.

Styling

For non-urgent events, the label of the left button is italic. For urgent events, as well as for non-event choices, the label uses a normal font.

Before each element a colored box is shown. The box is gray while that particular choice is not available. If choices are available, the box is green for controllable events, red for uncontrollable events, blue for other events, and for everything else (*time delay*, *reset*, and *undo*) it is black when using a light theme and white when using a dark theme. The text of the left button is colored similarly to the box. The colors help to quickly identify elements with possible choices.

Labels

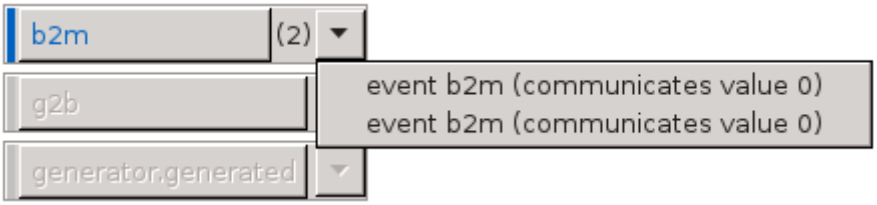
Between the buttons, a label may be shown, to provide more details. When the label is shown, and what text is shown, differs for the different choices. The sections below provide further details.

In the above image, the system has five events, besides the `tau` event. The simulator asks the user to choose between a transition for event `Switch.u_on` and a time transition.

Choosing an event transition

For an event, the first button makes it possible to quickly choose a transition for that event. If clicked, the possible transitions are filtered to those for the chosen event, and the automatic input component is asked to make the actual choice, based on its own configuration.

If multiple transitions are possible for a single event, the number of possible transitions is displayed as label between the two buttons for that event. The following image shows an example:

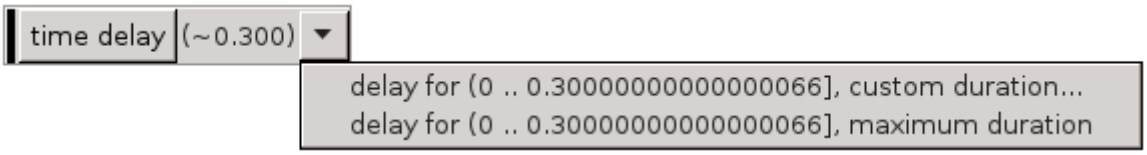


In this example, two transitions are possible for event `b2m`. By clicking the button labeled with `b2m`,

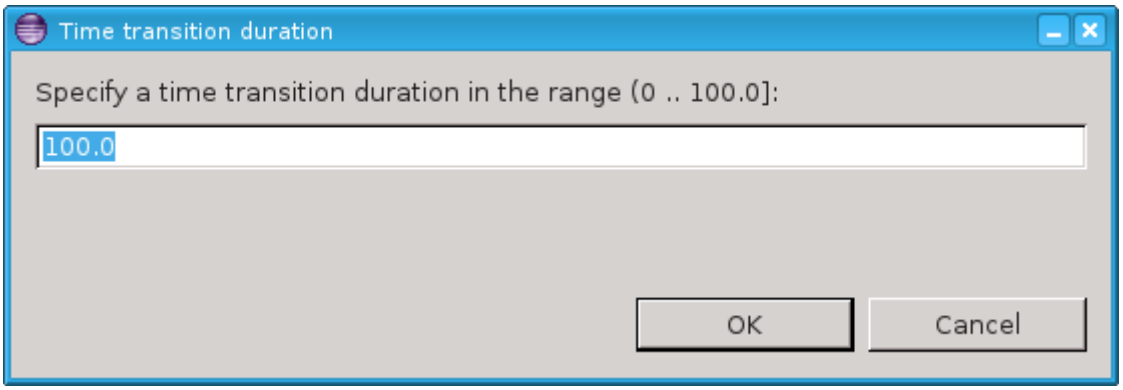
the automatic input component is asked to choose between the two transitions, based on its own configuration. By clicking the arrow button, a popup menu is shown, showing the two transitions. The popup menu allows manually choosing which transition to take, by clicking the menu item of the transition you wish to take.

Choosing a time transition and duration

The **[time delay]** button makes it possible to quickly choose the time transition. By default the maximum allowed delay duration will be used. This duration is displayed as label between the two buttons, in short form with three significant digits. By clicking the arrow button, additional options become available. These options also show the complete durations. The following image shows an example:



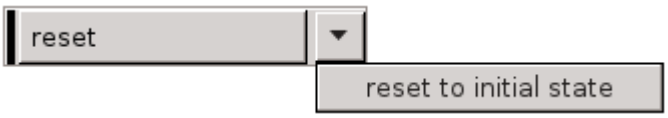
At most `0.30000000000000066` units of time can pass (`0.300` in short form). The first option is to choose a custom duration. The second option is to choose that maximum duration, similar to clicking the **[time delay]** button. The following image shows the dialog that is presented to the user to enter the custom duration:



By entering an integer or real number for the duration, that exact amount of time units will pass, after clicking the **[OK]** button. If the **[Cancel]** button is clicked instead, the choice is canceled, and the user is again free to choose between all available choices.

Choosing to reset the simulation

The **[reset]** button makes it possible to reset the simulation to the initial state. The popup menu that can be shown by clicking on the arrow button, has only one option, which the same effect as clicking the button. The following image shows the popup menu:

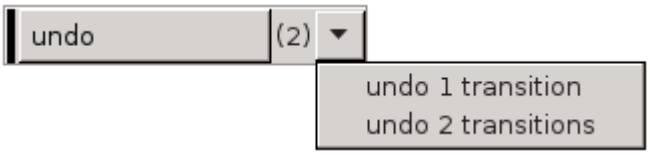


Choosing to undo some transitions

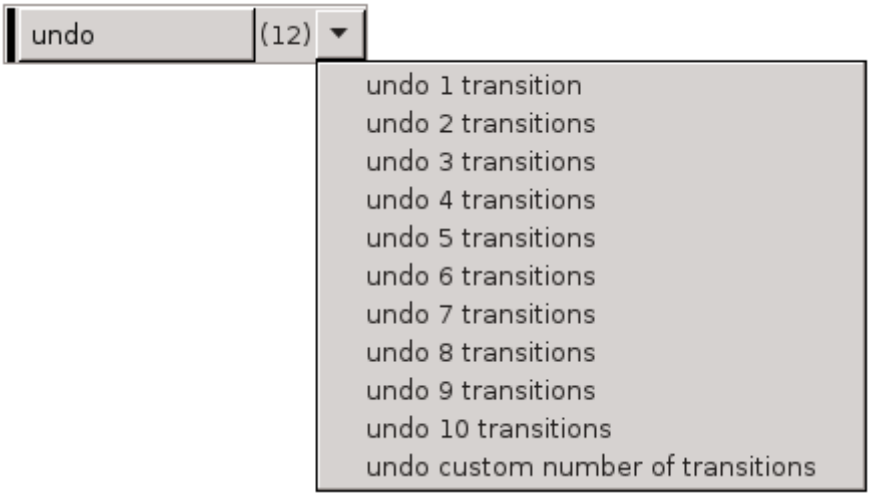
The **[undo]** button makes it possible to undo a single transition, and go back to the state just prior to that transition. The number of transitions that can be undone, is displayed as label between the

two buttons.

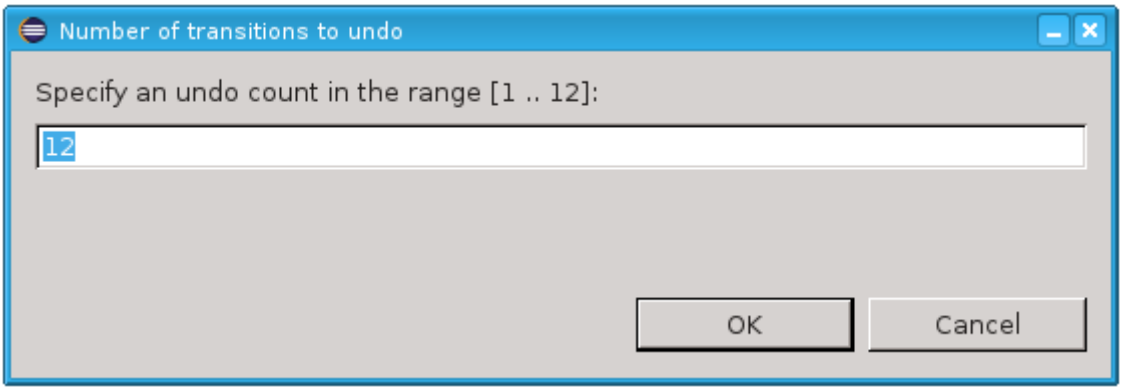
Clicking the arrow button shows a popup menu with additional options. Menu items are present to undo 1 transition, to undo 2 transitions, to undo 3 transitions, etc, for as many transitions as can be undone. The following image shows an example with two transitions that can be undone:



At most 10 such menu items are shown. If more than 10 transitions can be undone, an additional choice is present, that allows entering a custom number of transitions to undo. The following image shows an example with 12 transitions that can be undone:



Selecting that last option will present the user with a dialog, to select the number of transitions to undo:



The number of transitions needs to be entered as a positive number, within the given range. Clicking the **[OK]** button will undo the requested number of transitions. If the **[Cancel]** button is clicked instead, the choice is canceled, and the user is again free to choose between all available choices.

Semi-automatic mode

By default, in the interactive GUI input mode, the user is asked to choose, even if for instance there is only one possible transition. Having to choose for every transition can become tiresome. To reduce the number of situations where a choice is required, the semi-automatic mode is available.

Choices that are no longer available due to them being automatically chosen by enabling semi-automatic mode, are not shown in the GUI of the interactive GUI input mode. See also the Availability section above.

By enabling semi-automatic mode to automatically choose time transitions (option value `time`), while not automatically choosing the delay durations of time transitions (option value `timedur`), time transitions are automatically chosen, but the user is still asked to choose the duration of the time transitions. For this choice, all non-time choices are disabled, making only the time related choices available. The effect of the **[time delay]** button remains the same, i.e. choosing the maximum allowed or available delay. The popup menu for the time delay also still shows the default options, to choose either a specific custom delay duration, or the maximum duration.

By enabling semi-automatic mode to automatically choose the delay durations of time transitions (option value `timedur`), while not automatically choosing the time transition themselves (option value `time`), the effect of the **[time delay]** button changes from choosing the time transition with the maximum allowed or available delay, to choosing the time transition and letting the automatic input component choose the actual delay duration, based on its own configuration. The popup menu for the time delay then only shows one option, to let the automatic input component choose, identical to the effect of the **[time delay]** button.

Termination

By closing the GUI of the interactive GUI input component, no more interactive choices can be made. The next time an interactive choice is to be made, the simulation terminates.

The simulation can also be terminated by clicking the red stop button of the Eclipse console.

Automatic input mode

In the automatic input mode, the simulator automatically chooses transitions as well time transition durations. How the simulator chooses however, can be influenced using options. These options are explained below. The final section of this page explains the limitations of using these options.

Automatic mode choice algorithm option

In the automatic input mode, the simulator automatically chooses the transitions to take, without any user interaction. The choice that is made can however be influenced using the *Automatic mode choice algorithm* option (*Input* category). The following choices are available:

- First transition.
- Last transition.

- Random transition (random seed).
- Random transition (specific seed).

The options to choose the first or last transition do exactly what they state, i.e. they choose the first respectively the last transition from the list of possible transitions. Note that event transitions are always listed before time transitions, but event transitions can only be listed together with time transitions if non-urgent events are present. Also note that if transitions for the `tau` event are present, they are listed after the transitions for all other events.

The other two options randomly choose a transition from the possible transitions. If a random seed (for the random number generator) is used, each simulation will potentially choose a different trace. In this case, the seed that is used, is printed to the console at the start of the simulation. If a specific seed is used, then the choices are still random, but can be reproduced. Each simulation, using random automatic input mode, with the same seed, for the same specification, should lead to the same trace. Seeds are integer numbers from the range [0 .. 2^{30}].

Automatic mode time transition duration option

In the automatic input mode, the simulator automatically chooses the durations of the time transitions, without any user interaction. The choice that is made can however be influenced using the *Automatic mode time transition duration* option (*Input* category). The following choices are available:

- Maximum allowed duration.
- Random duration (random seed).
- Random duration (specific seed).

The option to choose the maximum allowed duration does exactly what it states, i.e. it chooses the maximum allowed duration for each time transition, until the next potential state event.

The other two options randomly choose a duration for the time transitions. For both options, a uniform random distribution is used, which is sampled to obtain the time at which time transitions are to be 'interrupted' or 'cut off'. The upper bound (exclusive) of the uniform random distribution is to be provided by the user. You can think of these random options as adding an extra automaton to the specification, that could look as follows:

```
automaton auto_time_trans_duration:
  const real UPPER = ...;
  disc dist real d = uniform(0.0, UPPER);
  disc real timeToWait;
  cont t der 1.0;

  location sampling:
    initial;
    edge do (timeToWait, d) := sample d goto waiting;

  location waiting:
```

```
    edge when t >= timeToWait do t := 0.0 goto sampling;
end
```

The upper bound of the uniform real distribution (`UPPER` > 0), is provided by the user. A sample is taken from the distribution, indicating the time to wait before the time transition is to be interrupted. Once that much time has elapsed, a new sample is taken, and the process repeats itself. The actual implementation is slightly different, in that no actual automaton is added, no additional `tau` transitions result from using this option, and no zero length time transitions are chosen. However, the effect is very similar.

So, say the user wants to use a uniform distribution over interval [0 .. 10). The simulator takes a sample, say 3.0. Assume we then get a time transition for at most 0.5 time units. We take that entire transition and are left with 2.5 remaining time units (3.0 - 0.5). If we then get another time transition for at most 1.2 time units, we take that one entirely as well. We are left with 1.3 (2.5 - 1.2) time units. If we would then get a time transition for at most 5.0 time units, the automatic input component ensures that we instead get a time transition for at most 1.3 time units, instead of one for at most 5.0 time units. Since we choose to interrupt after 1.3 time units anyway, there is no reason to compute the remainder of the time transition after those 1.3 time units. We choose the entire time transition of 1.3 time units, and compute a new sample. The simulator then once again calculates new transitions, and the process repeats. The next time transition that is calculated will be for the remaining 3.7 (5.0 - 1.3) time units, unless it is interrupted again before that time, i.e. if the new sample is smaller than 3.7.

If a random seed (for the random number generator) is used, each simulation will potentially choose a different trace. In this case, the seed that is used, is printed to the console at the start of the simulation. If a specific seed is used, then the choices are still random, but can be reproduced. Each simulation, using random automatic time transition duration, with the same seed, for the same specification, should lead to the same trace. Seeds are integer numbers from the range [0 .. 2^{30}].

The random durations are particularly useful in combination with non-urgent events, as it allows the non-urgent events to be chosen at the moment the time transition is interrupted. That is, it allows the non-urgent events to occur at random times. Using this option, the user can specify the distribution used for 'cutting up' the time transitions, and thus the possible moments when non-urgent events can potentially take place. This is in addition to the state events, at which the non-urgent events can also be chosen. These new additional possibilities however, are not influenced by the occurrence rate of the events of the system, and allow more control for the user. In particular for cases where the simulation allows infinite time transitions (no state events), the infinite time transitions can then be interrupted to allow non-urgent events to take place.

Limitations

Using the two options described above, and choosing random for each of them, introduces a certain degree of randomness to the trace that the simulator will explore. This randomness however, is rather limited. For one, the distributions that are used are mostly fixed, as the options don't allow using for instance an exponential distribution. Furthermore, there is no way to specify the distribution for events individually, and it is not possible to specify the distribution between event transitions and time transitions. Concretely, the options for instance don't enforce that non-urgent

events get chosen. They *may* be chosen when the time transitions are interrupted, or at state events, but that is not guaranteed, as the simulator may choose the time transition again, after the interruption.

The options provide a lightweight, quick, and easy to use approach to adding some randomness. If more control is needed, the environment can instead be modeled explicitly in the CIF model, thus defining when the non-urgent events may or will occur. One way to accomplish this, is by adding a use case.

Undo/reset

The automatic input mode does not support undo and reset.

Trace input mode

In the trace input mode, the simulator automatically chooses transitions as well as time transition durations, based on a user provided trace file.

Trace file

If the trace input mode is used, the *Trace input file* option (*Input* category) can be used to specify the path to the trace file. If the path is not explicitly specified, the input CIF file path is used, where the `.cif` file extension is removed if present, and a `.trace` file extension is added.

The trace file uses a line based file format. Leading (at the beginning of the line) and trailing (at the end of the line) whitespace is ignored. Empty lines as well as lines starting with a hash character (`#`) are ignored as well. Lines starting with a hash character can be used as comment lines. All other lines contain commands.

There are two types of commands: option commands and non-option commands. The option commands must always be before the non-option commands.

Option commands

All option commands use the `option NAME VALUE` syntax, where `NAME` is the name of the option, and `VALUE` is the value of the option. Option commands must be specified in the trace file, before any other (non-option) commands. The remainder of this section explains the available options.

- *option strict on/off* (default `off`)

  The strict option can be used to specify whether a strict match is expected for an event. If strict mode is on (option strict on), it is considered an error if more than one transition is possible for an event from the trace. If strict mode is off (option strict off), and there are multiple possible transitions for an event from the trace, the automatic input component is asked to choose between those possible transitions for that single event, based on its own configuration. The strict option applies only to events, not time delays.

- *option time off/implicit/explicit* (default `implicit`)

  The time option can be used to specify the time mode to use. If the time mode is off (option time off), time passage is not allowed. That is, time delays/transitions are never chosen, and the value

of variable time remains zero during the entire simulation. For this mode, the time command is not allowed.

If the time mode is implicit (option time implicit), time delays are chosen if necessary. The simulator will use the least amount of time passage possible, to still enable the events as specified in the trace. In other words, if an event is possible no time delays occur, and otherwise the simulator delays for as long as is needed until the event becomes enabled. For this mode, the time command is not allowed.

If the time mode is explicit (option time explicit), time delays are only possible if the time command explicitly allows time passage.

Transition commands

The transition commands can be used to specify the actual trace itself, to choose specific transitions. The following commands are available:

- *event NAME*

  The event command indicates that an event transition should be taken. The event to choose is indicated by the `NAME` part, which must be an absolute name of an event. For an automaton `a` with an event `e`, `NAME` must be `a.e`, and the full command would thus be event `a.e`.

  If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (e.g. `$plant`). For event commands however, no escape characters (`$`) should be used in event names.

  If the specified event is not enabled in the current state, the simulator checks whether time passage is allowed and possible. For the off time mode, time passage is not allowed, and the simulation results in deadlock. For the implicit time mode, the simulator delays if a time transition is possible, and otherwise simulation results in deadlock. For the explicit time mode, if the current event command was preceded by an explicit time command, the simulator delays if a time transition is possible, and otherwise (current command not preceded by time command or no time transition possible) simulation results in deadlock. Upon deadlock, simulation ends, and a warning is printed to the console to indicate the event that was not enabled.

  If multiple transitions are possible for the event, due to non-determinism, the strict option (see above) determines if and how the simulator proceeds.

- *input NAME = VALUE*

  The input command can be used to change the value of an input variable during a simulation. The input variable to assign is indicated by the `NAME` part, which must be an absolute name of an input variable.

  If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (e.g. `$plant`). For input commands however, no escape characters (`$`) should be used in input variable names.

  `VALUE` represents the new value assigned to the specified input variable. It must must match the

input variable's type. Only values of the following types are supported: booleans, integers, reals and strings. Only literal values may be used, so e.g. no `1 + 1`.

Assigning a value with a different type than the input variable's type results in a runtime error, causing the simulation to terminate. Assigning a value to an input variable that causes a state invariant to be violated, will lead to a deadlock and simulation will terminate.

- *time*

  The time command explicitly indicates that time is allowed to pass, i.e. the simulator may delay by choosing a time transition. This command is only allowed for the explicit time mode.

  The time command does not ensure that time passes. It specifies that time passage is allowed, not that it is required. Specifying the time command multiple times without an intermediate event command has the same effect as specifying the time command only once. That is, it doesn't matter how many times you allow time passage, once is enough.

The trace input mode does not support undo and reset.

Simulation

The commands from the trace file are processed in the order they occur in the trace file. After all commands have been used to select transitions, and no more commands are available, the simulation ends. The simulator indicates the simulation has terminated at the request of the user, as the user did not include any more commands in the trace file.

Simulation is also terminated if deadlock occurs or the user-specified simulation end time is reached, even if more (unprocessed) commands are still available.

SVG input mode

In the SVG input mode, SVG interaction is used. It allows SVG images that are used for SVG visualization to be used to interactively control the simulation. Since the SVG images are reused, using the SVG input mode requires the use of SVG visualization. If the CIF specification contains CIF/SVG input mappings, and SVG input mode is not used, the simulator simply ignores the input mappings.

This page explains the use of the SVG input mode of the CIF simulator at runtime (during simulation).

How it works

During simulation, as long as none of the interactive SVG elements (the SVG elements for which an input mapping exists) are clicked, the corresponding events or updates are never executed. For the other events (the non-interactive events), a choice is made using the automatic input component. The automatic input component makes a choice based on its own settings.

Hovering

When moving over one of the interactive SVG elements of the SVG visualization using the mouse pointer, the mouse pointer is changed and the outline of the SVG element is highlighted in red. This makes it easier to identify the interactive SVG elements in the SVG visualizer. When an interactive SVG element is clicked, the outline color is changed from red to green, for as long as the mouse button is not released.

The style for the outline of interactive elements can be changed using the *SVG input outline style* option (in the *Input* category). Specify `off` to not show any outline. Specify a hover color, click color and stroke width, separated by colons, to specify a specific style. For colors, there are two ways to specify them. The first way to specify a color is to use one of the named colors: `black`, `blue`, `cyan`, `dark-gray`, `gray`, `green`, `light-gray`, `magenta`, `orange`, `pink`, `red`, `white`, or `yellow`. The second way to specify a color is to indicate an RGB color, by specifying comma-separated red, green and blue components, each in range [0..255]. For the stroke width, specify the width in pixels as a positive floating point number. Note that if the element that gets an outline is scaled differently in the x and y directions, the stroke width may not be accurate. Some examples of valid option values are: `blue:magenta:2.0`, `255,0,128:black:0.5`. The default value is `red:green:1.5`.

The outlines, if not disabled using the *SVG input outline style* option (using its `off` value), can be selectively disabled for certain interactive SVG elements, using the *SVG input outline filters* option (in the *Input* category). By default, all interactive SVG elements get an outline when hovering or clicking them. By specifying filters, you can change the default. For this option, specify comma-separated node names of SVG elements for which to show the outline. Note that you must specify node names of the elements, as specified in the SVG standard. So, not the ids of the elements that you can specify yourself. For example, use option value `circle,rect,text` to only show outlines for circles, rectangles and texts. You can use `*` in a filter to match zero or more characters of an element name. Prefix a filter with `-` to exclude elements with matching node names, rather than including them. Initially, before applying any filters, all elements get an outline, as if `*,` if prefixed to the filters. The filters are applied in the order they are specified. As an example, specify `*,-text` or simply `-text`, to give all interactive elements an outline, except for SVG `text` elements.

Disabling the outline does exactly only that: it prevents the outlines from being shown. It does not prevent the mouse pointer from being changed. Also, interactive elements can still can be clicked and this will still have the same effect.

See the section on interactive groups for how hovering and outlines work for groups and elements of groups.

Clicking

The mouse click (and *not* the release) makes that the input component will interrupt any time transition that is currently being taken by the simulator. Then, when asked to choose a next transition, it will apply the input mapping for the interactive SVG element that was clicked. The event or update that results from the input mapping is then chosen. If the event is not enabled at that time, no transition is taken and a warning is printed to the console. To prevent such situations, you can use a monitor automaton. If multiple transitions are possible for an event, the choice (between the transitions for that event only) is deferred to the automatic input component, which makes a choice based on its own settings. Note that multiple transitions are only possible if

complete mode is enabled.

Queue

The SVG input mode allows interactively choosing certain events by clicking on interactive elements of an SVG image. Therefore, it is a semi-interactive input mode, as interactive inputs are combined with automatic behavior.

The image plays the part of the environment. Whenever an interactive element is clicked, the element is added to a queue. If a user quickly clicks the same element multiple times, or quickly clicks different elements, they are all stored in the queue, and the queue can grow. The transition to take is automatically chosen, based on the queue of interactive elements. If the queue is empty, the environment (the SVG image and the user) doesn't allow the associated interactive events. If the queue is not empty, the environment allows exactly one interactive event, the one matching the head of the queue. The model, together with the queue, form a closed model, without further interaction. Each interactive SVG element that is clicked, will be used in the order they were clicked, to choose transitions.

Reset the simulation

The simulation can be reset to its initial state. To do this, right-click on the SVG image and select **Reset simulation**.

Note that if the simulation has already terminated, it cannot be reset. A reset in the initial state is delayed until at least one other transition has taken place.

Unlike some other input modes, the SVG input mode does not provide a button or mechanism to undo a previously chosen transition.

Debugging

The console debugging output for CIF/SVG declarations includes debugging output for SVG input mappings. Input mappings that encounter runtime errors have stack traces similar to those of output mappings.

Responsiveness

SVG input is usually combined with real-time simulation. Clicking an interactive SVG element results in the application of an input mapping, which results in an event or update. That mapping is put in the queue. If a time transition is being taken by the simulator, the simulator will continue to take that time transition until the input component interrupts it. However, an input component can only interrupt a time transition during intermediate frames. As such, choosing a low frame rate (typically less than 10 frames per second) can significantly reduce the responsiveness of the SVG

input.

The **Reset simulation** menu item behaves like a queued click. It is processed by the simulator only when an intermediate frame allows the interruption of the current time transition, so its effect may be delayed if the frame rate is low.

Hovering interactive SVG elements highlights their borders, unless this is disabled. This too becomes less responsive when a low frame rate is used.

Interactive groups

It is possible to make a group of SVG elements (an `svg:g` element) an interactive element. By doing so, all the elements that are part of the group together become one single interactive element.

One practical example where this is very useful, is for buttons with a text label. In SVG, this requires the use of a rectangle and a text label. To ensure that clicking either one of them results in the same event or update, group the rectangle and text label together in a group. Then assign an id to the group and use that id in an input mapping. During simulation, when hovering over either the rectangle or the text label, by default both will be highlighted. Clicking on either of them results in the corresponding input mapping being applied, and the resulting event or update being chosen by the SVG input component.

In general, if an SVG image is clicked, the simulator will figure out on which graphical element the user clicked. If that element is itself an interactive SVG element, then the corresponding mapping is applied. If it is not an interactive SVG element, its parent (which may be a group) is checked. If the parent is not an interactive SVG element either, the parent of the parent is checked, etc. If none of the ancestors is an interactive SVG element, clicking the element has no effect on the simulation. Because of this behavior, it is strongly recommended not to make a child (or descendant in general) of an interactive SVG element interactive as well.

By default, when hovering an interactive group, all its direct non-group descendants are highlighted. For example, if an interactive group contains a rectangle and a text label, both the rectangle and the text label are highlighted by default. If an interactive group contains two sub-groups and both these sub-groups contain two rectangles, then all four rectangles are highlighted by default. You can configure which elements get highlighted using the *SVG input outline style* option. For example, for an interactive group representing a button, consisting of a rectangle and a text label, you can specify `*,-text` (or simply `-text`) to highlight the rectangle, but the text label. For more information about this option, see the section about hovering and outlines.

Semi-automatic mode

By default, in the interactive console and GUI input modes, the user is asked to choose, even if for instance there is only one possible transition. Having to choose for every transition can become tiresome. To reduce the number of situations where input is required, the semi-automatic mode is available. Semi-automatic mode allows the simulator to automatically choose certain transitions, while leaving the remaining choices for the user. The following choices can be automatically made by the simulator, in semi-automatic mode:

- Automatically choose a transition if there is only one transition possible.
- Automatically choose time transitions, regardless of how many transitions are possible.
- Automatically choose the delay duration of time transitions.
- Automatically choose event transitions (for certain events), regardless of how many transitions are possible.

Each of these possible automatic choices can be individually enabled, through the *Interactive mode automatic transition choice* option (*Input* category). As value for the option, comma separated *filters* should be supplied. The following filters are available:

- `1` to automatically choose a transition if there is only one transition possible.
- `time` to automatically choose time transitions.
- `timedur` to automatically choose the delay duration of time transitions.
- The name of an event, to automatically choose event transitions for that event.
- `urgent` to automatically choose event transitions for all urgent events.

For events, the absolute names of the events are used. That is, for an automaton `a`, with an event `e`, the absolute name of the event is `a.e`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in event names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. The special `urgent` event filter matches all urgent events of the specification. If an event filter doesn't match any of the events of the CIF model, a warning is printed to the console. If a filter doesn't have any effect, because a previous filter already turned ensured that the matching events into are automatically chosen, a warning is printed as well.

As an example, option value `1, g*, t*` consist of three filters: `1`, `g*`, and `t*`. The first filter indicates that the simulator should automatically choose a transition if there is only one transition possible. The second and third filters indicate that the simulator should also automatically choose all events starting with an `g` or a `t`, such as `g`, `g1`, `go`, etc. Note that the wildcards apply only to events, so `t*` does not mean that time transitions should be automatically chosen. The `t*` filter always matches the `tau` event, as that event is implicitly always present.

If an automatic choice can be made, given the filter and the possible transitions, the transitions are filtered to only keep the ones that can be chosen automatically, and they are sent to the automatic input component, which then makes the actual choice, depending on its own configuration. Similarly, if the delay duration of a time transition is to be automatically chosen, the automatic input component is asked to choose, based on its own configuration.

State history

The CIF simulator can be used to explore the state space of a CIF model. It is possible to see step by step, what is possible, and choose some transitions. Sometimes, after exploring a bit, you may want to go back a bit, to explore some alternative behavior. Obviously, you can terminate the simulation, and start a new one. There are several downsides to this. First, you have to start from the initial state again, while you may want to only go back a few states, and continue from there. Second,

restarting the simulator may take some time, especially for large models. Third, if you use several visualizations, you may have to layout them again, to be able to see all of them.

To make it easier to explore alternatives, the simulator supports resetting the simulation to the initial state, as well as undoing a certain number of transitions. This page explains that functionality in more detail.

Enabling history

The *History* option (*Input* category) can be used to enable or disable history. By default history is enabled. Using the option, it can be disabled. Disabling history disables both the reset and the undo functionality, which ensures no time or memory is consumed in keeping track of earlier states.

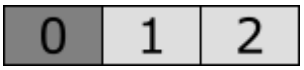
If history is enabled, it only has an effect if a (semi-)interactive input mode is used. Other (automatic) input modes will never perform a reset or undo. That is, the simulator will not keep track of any history states, unless an interactive (either a (semi-)interactive or semi-automatic) input mode is used, and history is enabled via the *History* option.

State stack

In order for the simulator to be able to go back to earlier states, it will need to remember those states. To that end, the simulator can store states that it has encountered in a state stack. The initial state is remembered separately. So, initially, the following is remembered:



Here, state `0` is the initial state. After two transitions are taken, the following is remembered:



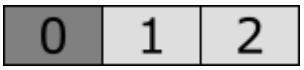
Here, two more states, numbered 1 and 2 are remembered. After one more transitions is taken, the following is remembered:



Here, one more state, numbered 3, is remembered. In each case, the last shown state is the current state. If we undo two transitions, the following is remembered:



As two transitions are undone, their resulting states (the last two states) are forgotten. If we take one more transition after the undo, we get the following:



This state 2 may be the same one as before, or it may be a new one, if a different simulation choice was made. If we reset the simulation, we forget all but the initial state, and end with up:



Stack size

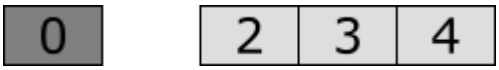
For long simulations, a lot of states will need to be remembered. Each state that needs to be remembered costs some amount of memory. To limit the amount of states that are remembered, the *History size* option (*Input* category) can be used.

If set to infinite, there is no limit to the amount of states that can be remembered. If set to a finite (non-negative) number, at most the given number of states are remembered, apart from the initial state. So, if set to zero, only the initial state is remembered. If set to one, one additional state is remembered, etc. The default history size is `100` states. Since if set to zero, only the initial state is remembered, this completely disables undo functionality.

By introducing a maximum number of states that can be remembered, this also introduces a maximum number of transitions that can be undone. Let's consider a history size of `3`. After three transitions, the following is remembered:



State `3` is the current state. It is possible to undo one, two, or three transitions, going back to states `2`, `1`, and `0`, respectively. Now consider what happens when we take another transition. We already remember three states besides the initial state, and have thus reached the maximum number of states to remember. By remembering the new target state of the fourth transition, we need to let go of state `1`, as it is the oldest remembered state, not taking into account the initial state. After this fourth transition, the following is remembered:



State `4` is the current state. We can undo one transition, to go back to state `3`. We can also undo two transitions, to go back to state `2`. However, as we forgot all about state `1`, we can't undo three transitions. Due to the *gap* between states `0` and `2`, we can't undo past state `2`. As you can see from this example, for a history size of 3, we can undo at most 3 transitions when there is no gap, and at most 2 transitions when there is a gap.

Availability

If history is disabled, reset is not available. If history is enabled, reset is available only if the current state is not the initial state. That is, reset is enabled for every state, except for the initial state.

If history is disabled, or the history size is zero, undo is not available. If history is enabled, and the history size is positive or infinite, undo is available. The number of transitions that can be undone depends on the number of states that is remembered. If `n` states are currently remembered, excluding the initial state, then `n - 1` transitions can be undone, at the least. If there is no gap, and the current state is not the initial state, then `n` transitions can be undone. Undo is only available in interactive or semi-automatic input modes. Undo is not available in the semi-interactive SVG input mode.

Requests

Reset and undo are only supported by some input modes, as explained above.

When the simulator interactively asks the user to choose between some possible transitions, it may also provide the possibility to reset the simulation to the initial state, or to undo some transitions. The details on how the different choices are presented, and how the choice can be made, are explained in the documentation for each of the interactive input modes.

If no transitions are possible, for instance due to deadlock or reaching the end-user provided simulation end time, the simulator may interactively ask the user to choose between termination of the simulation, resetting the simulation to the initial state, or undoing some transitions. This choice is only presented if either reset or undo is enabled, as otherwise there is no choice to make. The details on how the different choices are presented, and how the choice can be made, are explained in the documentation for each of the interactive input modes.

Semi-automatic mode

By using semi-automatic mode, a purely interactive input mode, can be made semi-automatic. This can lead to unexpected results, in combination with reset and undo.

Consider a situation where four transitions have been taken. The first transition was manually chosen, as was the third. The second and fourth transition were automatically chosen. The user is asked to interactively choose the fifth transition. Consider what happens if the user chooses to undo three transitions. The simulator goes back to the state reached after the first transition. The possible transitions are the same as the first time they were calculated. An automatic choice was made for the second transition, so one will also be made the second time around. So, after the second transition is once again automatically chosen, the user is interactively asked to choose the third transition. So, instead of undoing three transitions and getting a choice for the second transition, three transitions are undone, a second transition is automatically chosen, and the user is asked to choose for the third transition.

Similarly, undoing the fourth transition, will result in a fourth transition being automatically chosen again. If the same choice is made as before, undoing the fourth transition has no effect.

Resetting the simulation to the initial state, will not lead to the user being able to choose the transition to take from the initial state, if the first transition is automatically chosen.

Non-urgent events

By default, the CIF simulator treats all events as urgent. That is, whenever a transition is possible for any event, time may not progress. In other words, events take priority over time passage. However, not all events should be urgent. As such, the simulator has a *Non-urgent events* option (*Input* category), that makes it possible to let the simulator know about the *non-urgent events*. An example of a non-urgent event is a button that can be pushed. The button can always be pushed, now or in the future. The corresponding event is thus always enabled, but time should still be able to progress, as the event may not actually happen at this time.

As value for the *Non-urgent events* option, comma separated *filters* should be supplied. Each filter specifies one or more non-urgent events. The absolute names of the events are used. That is, for an automaton `a`, with an event `e`, the absolute name of the event is `a.e`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in event names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. The special `svg` filter can be used to turn all SVG

interactive/input events into non-urgent events. If a filter doesn't match any of the events of the CIF model, a warning is printed to the console. If a filter doesn't have any effect, because a previous filter already turned the matching events into non-urgent events, a warning is printed as well.

As an example, option value `e, a.e, f*, t*` consists of four filters: `e`, `a.e`, `f*`, and `t*`. The first filter indicates that event `e` is a non-urgent event. The second filter indicates that event `e` in automaton `a` is a non-urgent event. The third filter indicates that all events whose absolute name starts with an `f` are non-urgent events. This could include events `f`, `f1`, `foo`, `f.e`, etc. The fourth filter is similar to the third filter, but uses character `t` as a prefix, rather than character `f`. The `t*` filter always matches the `tau` event, as that event is implicitly always present.

Simulating events as non-urgent is not supported when the execution mode is enabled.

Input modes

By default, the simulator does not know which events of the model are non-urgent events. By letting the simulator know about them, it can properly simulate them with non-urgent semantics. The CIF simulator however, has several input modes. This section discusses for each input mode, the influence of non-urgent events, and their practical use.

- Interactive console input mode

  When using the interactive console input mode, the simulator will interactively ask the user to choose transitions, from the list of possible transitions printed to the console.

  If however the CIF model contains events that are always enabled (such as a button that can always be pushed), time may never progress. The simulator treats all events as urgent. By specifying the always enabled events as non-urgent events, the user is free to choose between those non-urgent events and time passage (time transitions). If combined with the semi-automatic mode for the urgent events, the user can choose the non-urgent events and time transitions, while the other/urgent events are automatically chosen by the simulator, reducing the number of choices the user has to make.

- Interactive GUI input mode

  When using the interactive GUI input mode, the simulator will interactively ask the user to choose transitions, using a GUI (Graphical User Interface).

  If however the CIF model contains events that are always enabled (such as a button that can always be pushed), time may never progress. The simulator treats all events as urgent. By specifying the always enabled events as non-urgent events, the user is free to choose between those non-urgent events and time passage (time transitions). If combined with the semi-automatic mode for the urgent events, the user can choose the non-urgent events and time transitions, while the other/urgent events are automatically chosen by the simulator, reducing the number of choices the user has to make.

- Automatic input mode

  When using the automatic input mode, the simulator will automatically choose transitions,

based on certain criteria. If the CIF model contains a use case, the simulator automatically simulates that use case.

If however the CIF model contains events that are always enabled (such as a button that can always be pushed), time may never progress. The simulator treats all events as urgent. By specifying the always enabled events as non-urgent events, the simulator is free to choose between those non-urgent events and time passage (time transitions). If combined with the *random* automatic mode choice algorithm, and *random* automatic mode time transition duration, this allows for the automatic validation of arbitrary traces.

- SVG input mode

  When using the SVG input mode, the simulator will automatically choose transitions, based on certain criteria. The non-urgent events are usually coupled to the SVG image (using CIF/SVG input mappings). This ensures that the non-urgent events are only enabled when the corresponding interactive element of the image is clicked by the user. The SVG image plays the role of the environment, closing the system. If the SVG image only partially closes the system, the remaining non-urgent events can be specified, similar to the way they are specified for the automatic input mode.

Supervisory control application domain

This section discusses non-urgent events in the application domain of supervisory control, where the system consists of a plant (the hardware), and a supervisor (or other controller that restricts the plant behavior).

Informally, we can distinguish the following types of events:

- *Controller events*

  Controller events are those events that are initiated by a controller, such as a supervisor. These events are usually controllable events. However, if supervisor synthesis is not used, the events need not necessarily be controllable.

  For controller events, the behavior is usually fully specified in the CIF model. That is, if the CIF model contains both the plant and the controller, the CIF model determines when the events are enabled, and when they are disabled. The controller events should happen as soon as they are enabled. That is, once the controller determines that for instance a motor should be turned on, the motor should be turned on without further delays. Thus, controller events are urgent.

  For low-level controllers, the events generally correspond directly to the events of the actuators. For instance, a controller may turn a lamp on or off. For higher level controllers however, there usually is no such direct relation. Furthermore, internal events of the controller (usually event `tau`, or other events that are neither controllable nor uncontrollable), are considered controller events as well.

- *Plant events*

  Plant events are those events that are initiated by the plants. These events are usually

uncontrollable events. However, if supervisor synthesis is not used, the events need not necessarily be uncontrollable.

For plant events, the behavior is usually fully specified in the CIF model. The CIF model models exactly when the events become enabled. For instance, when an elevator moves and reaches its top position, the event that indicates that 'the top position sensor goes on' becomes enabled, and should be executed at that time. That is, the plant model models/follows the physical behavior, and determines when the plants events are enabled. This behavior is fully specified and known, and further delays are undesirable. As such, plant events are urgent.

For low-level plants, the events generally correspond directly to the events of sensors. Typical examples include limit sensors, such as the elevator top position sensor described above, and sensors that detect products throughout the system. For higher level plants however, there usually is no such direct relation. Furthermore, internal events of the plants (usually event `tau`, or other events that are neither controllable nor uncontrollable), may be considered plant events as well.

- *Environment events*

  Environment events are those events that are initiated by the environment (anything outside and the plants and the controller). These events are usually uncontrollable events. However, if supervisor synthesis is not used, the events need not necessarily be uncontrollable.

  The CIF model only observes or monitors the environment events. The environment (outside of the behavior specified in the CIF model), determines when, if ever, the events actually happen, and thus defines their behavior. From the point of view of the CIF model, the environment events can happen at any time, and as such the CIF model allows them at all times. The behavior is left completely unrestricted/free in the CIF specification, as it is unknown when the events will actually happen. Since time may pass before these events actually happen (as determined by the environment, rather than the CIF model), these events are thus non-urgent (from the point of view of the CIF model). If the CIF model would instead consider them to be urgent, the model would always forbid time to progress, even if the environment does not currently allow any environment event to occur, leading to deadlock.

  For low-level plants, the events generally correspond directly to the events of sensors. For instance, the buttons of a user interface can be pushed and released. Other typical examples of environment events include events for other user interface elements, and events that indicate that a product has entered the system, from the environment of the system. For higher level plants however, there usually is no such direct relation. Furthermore, internal events of the plants (usually event `tau`, or other events that are neither controllable nor uncontrollable), may be considered plant events as well.

Thus, for plant events as well as for controller events, the CIF model specifies/defines when the events may/should occur, while for environment events the specification always allows them, and lets the environment choose/determine when they actually occur. In other words, plant/controller events originate from within the specification (internal behavior), while environment events originate outside of the specification (external behavior).

A closed system is a system for which the entire behavior is specified in the CIF model. Closed

systems have no environment events.

An open system does not model the environment. As such, open systems typically have plant events, controller events, and environment events. The environment events are then unrestricted in the CIF model, leaving the environment (which is not specified) free to determine its own behavior, and thus the behavior (enabledness) of the environment events.

An open system can be closed by adding a use case, which models one possible sequence of behavior of the environment, over time. That is, the added use case eliminates environment events, by defining when they are enabled. Open systems can also be closed by the simulator, as described above for the different input modes.

As an example, consider the following open system:

```
plant automaton Button:
  uncontrollable u_pushed, u_released;

  location Released:
    initial;
    edge u_pushed goto Pushed;

  location Pushed:
    edge u_released goto Released;
end

plant automaton Lamp:
  controllable c_on, c_off;

  location Off:
    initial;
    edge c_on goto On;

  location On:
    edge c_off goto Off;
end

supervisor automaton ButtonLampController:
  location:
    initial;
    edge Lamp.c_on  when Button.Pushed;
    edge Lamp.c_off when Button.Released;
end
```

Here we have a simple button and a lamp. The lamp can only be turned on while the button is pushed, and will be turned off as soon as the button is released. The button events are environment events, since it always possible to either push the button, or release it.

By specifying the button events as non-urgent events, and specifying the lamp events as semi-automatic events, we can use the interactive console input mode or the interactive GUI input mode

to try different traces, where we only have to choose between the button events and how much time we want to let pass.

Alternatively, we could add a use case, modeling one possible behavior of a user:

```
plant automaton User:
  cont t der 1.0;

  location:
    initial;
    edge Button.u_pushed, Button.u_released when t >= 1 do t := 0.0;
end
```

This specifies that the user either pushes or releases the button, after one unit of time has passed (typically one second). Since clock `t` is reset each time, the user repeats this behavior. Thus, the button is pushed after one second, for one second, after three seconds, for one second, after five seconds, for one second, etc. This extended model (with use case), can be simulated using the automatic input mode, as the `User` plant automaton closes the system.

Simulation output

The CIF simulator can produce various forms of output.

Console output

The default form of output is to the console. The type of console output to produce, is determined by the *Output mode* option (*General* category). The following modes are available:

- *Error*: produces only error output. It is recommended not to use this mode. Use the warning mode instead, if you wish to limit the amount of console output.
- *Warning*: produces error and warning output. Use this mode to be informed about potential problems, while keeping the amount of console output to a minimum.
- *Normal*: produces error, warning, and normal output. This is the default output mode. The information that is printed to the console for normal output can be configured using the Normal output option.
- *Debug*: produces error, warning, normal, and debug output. This mode can be used to print additional debugging information to the console. The information that is printed to the console for normal output can be configured using the Debug output option.

The simulator can thus produce two types of configurable console output:

- Normal console output
- Debug console output

Output components

Besides console output, the simulator also features a powerful and extensible output framework. This framework allows for the addition of output components, such as visualizers. The following output components are currently available:

- Print output
- SVG visualizer
- Trajectory data output
- Plot visualizer
- State visualizer

Other topics

- Real-time simulation

Normal console output

Normal console output mainly focuses on the state of the specification, and the progress of the simulation.

Available types of normal output

The following types of normal output are available:

- *The initial state* [`state-init`] (default)

  Prints the initial state to the console.

- *The target states* [`state-target`] (default)

  Prints all target states to the console. This includes all states reached by event transitions and time transitions. Specifically, it excludes the initial state, and it includes the final/deadlock state.

- *The final state* [`state-final`]

  Prints the final state to the console, regardless of whether or not the final state is a deadlock state.

- *The deadlock state* [`state-deadlock`]

  Prints the final state to the console, but only if it is a deadlock state.

- *The intermediate states* [`state-intermediate`]

  Prints all intermediate states to the console. Intermediate states are the states that are used to generate intermediate frames, for smooth visualizations. The intermediate states are generated for states between the start and end states of time transitions, when real-time simulation is enabled. Note that this does not include the initial state, target states, and final/deadlock states.

- *The algebraic variables as part of the state* [`state-alg-vars`] (default)

  Prints the algebraic variables (derived state) as part of the state. This applies to all states, including the states for intermediate frames. If no state information is printed, this output type has no effect. Disabling printing of algebraic variables excludes algebraic variables, regardless of whether state filtering includes or excludes them.

- *The derivatives as part of the state* [`state-derivs`] (default)

  Prints the derivatives of the continuous variables (derived state) as part of the state. This applies to all states, including the states for intermediate frames. If no state information is printed, this output type has no effect. The derivative of variable `time` is not included. Disabling printing of derivatives excludes derivatives, regardless of whether state filtering includes or excludes them.

- *The possible transitions (for interactive console choice only)* [`trans-minimal`]

  Prints the possible transitions to the console, if the interactive console input mode is used. Multiple possible transition are always printed. Single possible transitions are only printed if automatic choice for a single possible transition is disabled. Nothing is printed if no transitions are possible.

- *The possible transitions (for interactive console choice, or if more than one transition)* [`trans-default`]

  Similar to 'interactive console choice only', but additionally always prints the possible transitions, if there are multiple possible transitions, regardless of the input mode. If complete mode is enabled, then there may be cases where not all possible transitions are calculated, and thus also not all possible transitions are printed.

- *The possible transitions (always)* [`trans-always`]

  Always print the possible transitions to the console, regardless of the input mode, and the number of possible transitions. However, nothing is printed if no transitions are possible. Also, if complete mode is enabled, then there may be cases where not all possible transitions are calculated, and thus also not all possible transitions are printed.

- *The chosen transitions* [`chosen-trans`] (default)

  Prints the chosen transitions (the transitions that are about to be taken), to the console.

- *The interrupted transitions* [`interrupted-trans`] (default)

  Prints information about interrupted transitions (time transitions that are stopped prematurely) to the console.

- *The simulation result* [`sim-rslt`] (default)

  Prints the simulation result the console, at the end of the simulation, if simulation did not result in a runtime error.

- *The random seeds used for the random generators* [`seeds`] (default)

  Prints the seeds used for the random generators to the console, to allow reproducing the simulation trace. Only if a random seed is used, is the seed printed. That is, for user-specified seeds, the seed is not printed. The following seeds may be printed:

  - The seed used for the random automatic mode choice algorithm.
  - The seed used for the random automatic mode time transition duration.
  - The seed used for the first random generator for a stochastic distribution. The seed is only printed if it is used to create a random number generator for at least one stochastic distribution.

- *The print declarations output* [`print`] (default)

  Prints the output generated by print declarations.

Normal output option

In the option dialog, each of the different types of output can be enabled and disabled individually (*Normal output* option, *Output* category).

From the command line, using the `-o` or `--output` option, the names of the different types of normal output, as specified between square brackets, should be used, separated by commas. The output types indicated with '(default)' are enabled when default settings are used, i.e. when the `-o` or `--output` option is not used. As an example, consider `-o print,seeds` or `--output=print,seeds`. Only output generated by print declarations (due to `print`) and output about the used seeds for random generators (due to `seeds`) is enabled, while all other output is disabled.

The output types that are specified using the option replace the default output types. However, it is also possible to specify additions and removals relative to the default output, by prefixing output types with a `+` or `-` respectively. For instance, when using `-o -print,+trans-always` or `--output=-print,+trans-always` the default output is used, with print declarations output excluded from it (due to `-print`), and with always printing possible transitions enabled (due to `+trans-always`). Replacements (no prefix) may not be combined with additions/removals (`+` or `-` prefix).

Specifying a replacement output type twice leads to a warning being printed to the console. Adding an output type that is already present or removing an output type that is not present, also leads to a warning being printed.

Output details

When printing states, variable `time` is always printed first. It is followed by the current location of each of the automata, the current values of the state variables (the discrete, input and continuous variables), the current values of the derivatives of the continuous variables, and the current values

of the algebraic variables. Except for variable `time`, the state objects are sorted alphabetically based on their absolute names. For automata with a single nameless location, the current location is always `*`.

For variables with a function type and implicit default values, the default values may include `defaultValueFunc`, `defaultValueFunc2`, etc.

When printing the possible transitions, the event transitions are printed first, followed by the `tau` transitions, and the time transition. The event transitions are sorted alphabetically on the names of the events.

State filtering

By default, when printing states, all state objects are included. Using the *Normal output state filters* option (*Output* category), the state objects can be filtered. The *Normal output state filters* option only has effect if states are printed at all.

As value for the option, comma separated *filters* should be supplied. Each filter specifies one or more state objects. The absolute names of the objects are used. That is, for an automaton `a`, with a variable `x`, the absolute name of the variable object is `a.x`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in the names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. If a filter doesn't match any of the state objects of the CIF model, a warning is printed to the console.

By default, filters include matching state objects. Filters may however be preceded by a `-` character, turning them into exclusion filters, which exclude matching states objects rather than including them. Filters are processed in the order they are specified, allowing for alternating additions and removals. If a filter does not result in the addition/removal of any state objects to/from the filter result, a warning is printed to the console. A warning is also printed if the entire state is filtered out.

As an example, option value `a.*,-a.b*,a.bc*` consists of three filters: `a.*`, `-a.b*`, and `a.bc*`. The first filter indicates that state objects whose absolute names start with `a.` are to be included. The second filter indicates that from those matching state objects, the state objects whose absolute names start with `a.b*` are to be excluded. To that result, the third filter adds those state objects whose absolute names start with `a.bc*`. For instance, if a specification contains state objects `time`, `a.a`, `a.b` `a.bb`, `a.bc`, `a.b.c`, `a.bc`, `a.bcc`, and `a.bcd`, the result of the three filters is that the following state objects are displayed: `a.a`, `a.bc`, `a.bc`, `a.bcc`, and `a.bcd`.

The default option value (filter) is `*`.

Regardless of whether the filtered result includes them or not, all algebraic variables and derivatives of continuous variables may be excluded, using the Normal output option.

Profiling

If profiling is enabled, all normal output is automatically disabled. That is, profiling suppresses all normal output.

Debug console output

The following types of debug output are available:

- *Debug the CIF/SVG declarations* [`svg`]

  Prints debugging information to the console, for the CIF/SVG declarations. For more information on this kind of debugging output, see the page on console debugging for CIF/SVG declarations.

- *Debug the CIF/SVG real-time timing* [`svg-timing`]

  Prints debugging information to the console, for the CIF/SVG real-time timing. For more information on this kind of debugging output, see the page on console debugging for CIF/SVG real-time timing.

- *Debug the parser* [`parser`]

  Prints debugging information to the console, for the parser. This information is primarily intended for the developers of the simulator, and not for end users.

- *Debug the generated code (by writing it to disk)* [`gen-code`]

  The simulator generates highly optimized code for the specification. Enabling this debugging output, writes the generated code to disk, for further debugging. This information is primarily intended for the developers of the simulator, and not for end users.

- *Debug the ODE solver* [`ode`]

  Prints debugging information to the console, for the ODE solver. This information can for instance be used to figure out why a state event (guard change) was missed.

Debug output option

In the option dialog, each of the different types of output can be enabled and disabled individually (*Debug output* option, *Output* category).

From the command line, using the `-d` or `--debug` option, the names of the different types of debug output, as specified between square brackets, should be used, separated by commas. By default, i.e. when not using the `-d` or `--debug` option, all debug output is disabled. As an example, consider `-d parser,svg` or `--debug=parser,svg`. Debug output is printed for the parser (due to `parser`) and CIF/SVG declarations (due to `svg`).

Specifying an output type twice leads to a warning being printed to the console.

Enabling any debug output, automatically changes the output mode to 'debug mode'.

Profiling

If profiling is enabled, all debug output is automatically disabled. That is, profiling suppresses all debug output.

Print output

The CIF simulator supports several forms of output. By default, simulation results, such as the current state, are printed to the console. While this provides useful information, it can also be useful to generate custom output, tailored to one's specific needs. To allow this, the CIF simulator supports print output. Print output, by means of print declarations in the CIF specification, allows to print custom text for certain transitions that occur during simulation, either to the console or to a file.

Print output can be used to generate text that after simulation is further processed, for instance for visualization or for analysis using statistical software. Print output can also be useful for debugging.

Here we discuss the following topics related to simulation of CIF specifications with print declarations.

- Overwriting files
- Single open
- Unused print file declarations
- Line-based output
- Output order
- Enabling/disabling print output
- Undo/reset

Overwriting files

If output is printed to files, those files are always overwritten for each simulation. That is, at the start of the simulation, the output file is removed if present. A new empty file is then created, to which the output is printed during the simulation.

Single open

If the same file or special target is used/specified multiple times, in different print file declarations or locally in print declarations, the text produced by all these print declarations is all printed to that single file or special target. The simulator only opens a stream to the file or special target once, and prints to it all the output of all print declarations that apply to that file or special target.

Unused print file declarations

If a print file declaration does not apply to any print declarations, it is unused. The CIF simulator will still open the file or special target for writing, but no text will be printed to it. A warning is printed to the console, to indicate the potential problem. However, no such warnings are ever printed for special target 'stdout'.

Line-based output

Output is printed in a line-based fashion. That is, every time text is printed for a print declaration, the text is printed on a new line. If a single print declaration has two texts, one for the pre/source state and one for the post/target state, both texts are printed on new lines.

Output order

The simulator guarantees that once text is printed for a certain state, no text shall ever be printed during that same simulation, for previous states. That is, output text is ordered by 'increasing' states. This guarantee holds for file output. For the special targets standard output stream (stdout) and standard error stream (stderr), the output may get mixed, for instance on the Eclipse ESCET IDE's console.

The order in which text is printed for different print declarations that print text for the same state, is determined by the order in which the print declarations are specified in the CIF specification, from top to bottom. The order is determined per file or special target, after elimination of imports, component definition/instantiation, etc. A depth-first traversal of the component hierarchy is used, where the print declarations of components are considered before the print declarations of their children.

Enabling/disabling print output

Print output is enabled by default. It can be disabled by reducing the types of normal console output using the *Normal output* option (*Output* category). Using that option it is also possible to disable all other forms of normal output, to be able to see only the print output on the console. Print output can also be disabled by changing the output mode to *Error* or to *Warning*, instead of *Normal*.

Undo/reset

By undoing one or more transitions, or by resetting the simulation, the state may change. However, only previously taken transitions are undone. No new transitions are taken by undoing transitions, or by resetting the simulation. As such, no new print output is generated.

SVG visualizer

The CIF simulator supports several forms of output. By default, simulation results, such as the current state, are printed to the console. While this provides useful information, it is not easy to decipher the precise state of a (sub-)system. Also, it does not give a graphical overview of the current state of the system.

The simulator however, also supports SVG visualization, which allows the user to make an image of the system, and have that image be updated by the simulator during simulation, based on the current state of the system. This allows the user to see changes in the state of the system, in a graphical representation of that system. This makes it much easier to get a global overview of the system during simulation. Since the user makes the image, the representation directly corresponds to the way the user sees the system.

The following topics with further information are available:

- Simulation of SVG visualizations
- Debugging SVG visualizations
- SVG viewer and visualizer

For SVG visualization with interaction, see the SVG input mode.

Simulation of SVG visualizations

A CIF specification with CIF/SVG declarations can be simulated, and the connected SVG image or images are then visualized. This page discusses the following simulation related topics:

- Batik rendering library
- Unsupported SVG features
- Visualizer name
- Unused images
- Application order
- SVG visualization options
- Demonstrations
- Undo/reset

Batik rendering library

The CIF simulator uses the free and open source Apache Batik SVG Toolkit to render SVG images. During initialization, the CIF simulator loads the SVG image file or files that the user provided, and subsequently has access to their XML trees. It then uses the Batik library to draw the SVG images represented by the XML trees.

Unsupported SVG features

Multiple versions of the SVG standard exist. The above-mentioned Batik library supports version 1.1 of the SVG standard. Using an SVG image with unsupported SVG features results in the following error when the SVG file is opened in the ESCET IDE:

```
ERROR: SVG image file "some_image.svg" is not an SVG file, is an invalid SVG file, or
contains unsupported SVG features.
CAUSE: invalid.element
```

Visualizer name

For each SVG image that is used in the CIF model, an SVG visualizer is opened by the simulator. The SVG visualizer is given the name of the corresponding SVG file, without its path and its `.svg` file extension.

Unused images

If an SVG image is connected to a CIF model, but there are no CIF/SVG input or output mappings connected to it, the image is not used to represent the state of the model, nor to control the model through interaction. In such cases, the simulator prints a warning to the console. Note that such warnings are printed even if a connected image has SVG copy and/or move declarations associated to it, but no input or output mappings.

Application order

When the simulator is started, the SVG image is loaded. First, the CIF/SVG copy declarations are applied, to ensure all elements exist. For copies of copied elements, the former are applied after the latter, to ensure the elements that need to be copied actually exist. Besides that, the copy declarations from a CIF file are applied in the order in which they are specified (after elimination of component definition/instantiation). The copy declarations are only applied once.

After the copy declarations, the CIF/SVG move declarations are applied. The move declarations from a CIF file are applied in the order in which they are specified (after elimination of component definition/instantiation). The move declarations are only applied once.

The CIF/SVG output mappings are applied after that, for the initial state. After the output mappings have been applied, the resulting SVG image is shown by the simulator. The output mappings are applied again and again, as the state of the CIF model changes.

The output mappings from a CIF file are applied in the order in which they are specified (after elimination of component definition/instantiation). Note however, that the order does not matter, as all output mappings are unique and independent of each other.

Each mapping that is applied, results in an update of an attribute or text label. Either the new value is different from the previous value, or the value is updated to the value that was already there. The results of the mappings are cumulative. That is, the first mapping is applied to the original SVG

image, the second mapping is applied to the image that resulted from applying the first mapping, the third mapping is applied to the image that resulted from applying the second mapping, etc.

During simulation, the simulator provides states to the SVG visualizer, which applies the mappings to each of those states, in order to generate frames. Whenever the mappings are applied, they are all applied, one after the other. Each mapping is applied once per frame. Between the application of the individual output mappings, the display is never updated. That is, only after all mappings are applied, is the visualization updated. This assures that the images are in a consistent state when they are rendered.

By changing the `width` and `height` attributes of the root `svg` element of an SVG image, it is possible to change the size of the image, and thus the canvas on which it is displayed during simulation. Changes to these attributes currently only have effect for the initial state. If the size of the image is changed during simulation, such changes are ignored.

SVG visualization options

The *SVG visualization* option (*Output: SVG visualization* category) can be used to disable SVG visualization. By default, the option is enabled, and SVG visualization is used if the CIF file being simulated declares any CIF/SVG declarations. By disabling the option, SVG visualization is disabled, regardless of whether the CIF specification has any CIF/SVG declarations.

When simulating with SVG visualization, it is often useful to set a frame rate to enable real-time simulation.

Demonstrations

For demonstrations and presentations, it can be useful to record a video of a simulation with SVG visualization. This removes the need to manually control the simulation during the presentation. Furthermore, a pre-recorded video can't fail to execute, which can be a risk during live demonstrations. Finally, a pre-recorded video doesn't change, and is not influenced by the performance of the computer on which the demonstration is given.

To record a video of the SVG visualization, screen recording or screen capturing software can be used. For instance:

- Snipping Tool for Windows (`Win` + `Shift` + `S`)
- Screenshot for Mac (`Shift` + `Command` + `5`)
- recordMyDesktop for Linux.

Ideally a capturing software supports capturing a region of the screen, so that it is for instance possible to only capture the visualization. Furthermore, it may be beneficial to be able to capture the mouse pointer, so that it is clear where interaction takes place, etc.

Undo/reset

By undoing one or more transitions, or by resetting the simulation, the state may change. The SVG visualizer is automatically updated to reflect such changes.

Debugging SVG visualizations

If during a simulation an SVG visualization does not behave as you would expect it to behave, there are several ways to debug such problems.

The following approaches are available to debug behavioral issues:

- Console debugging for CIF/SVG declarations
- Save as SVG
- Stack traces

The following approaches are available to debug timing/smoothness issues:

- Console debugging for CIF/SVG real-time timing

Console debugging for CIF/SVG declarations

Console debugging output for CIF/SVG declarations can be enabled or disabled by using the *Debug output* option (*Output* category). It is disabled by default. If enabled, it produces debug output to the console, whenever CIF/SVG declarations are applied. Here is an example of what such output looks like:

```
SVG output ("some_file.svg") id "a" text: "10.00"
SVG output ("some_file.svg") id "b" text: "0.0"
SVG output ("some_file.svg") id "c" text: "3.16"
SVG output ("some_file.svg") id "n" attr "fill" (CSS attr): "#ff0000"
SVG output ("some_file.svg") id "water" attr "height" (SVG attr): "75.0"
SVG output ("some_file.svg") id "water" attr "y" (SVG attr): "104.5"
```

Each of the output mappings that is applied, results in a line of debugging output. For output mappings that update the value of an attribute, the output indicates whether the attribute is handled as an SVG presentation attribute or as a CSS style attribute.

The textual representations of the values that are set, are put between double quotes, and their content is not escaped in any way.

Due to the asynchronous nature of the SVG visualizer, the simulator will continue to calculate the next states that are to become a frame, while the SVG visualizer is still applying mappings for previous frames. As such, the debug output and other simulation output are likely to be out of sync and interleaved. To ease debugging, you can enable the test mode option, which ensures synchronous execution, and thus synchronous output.

For the input mappings, whenever an interactive SVG element is clicked, and the corresponding mapping is used to determine the event or update to take, a line of debug output is produced. Here is an example of what such output looks like:

```
SVG input ("some_file.svg") id "button": event "button.u_pushed"
Transition: event button.u_pushed
```

From the line of debugging output, it can be seen that the SVG element with id `button` was clicked, and this resulted in the execution of event `button.u_pushed`.

Similar debug output is available for the other CIF/SVG declarations:

```
SVG copy ("some_file.svg") id "button" post "1".
SVG copy ("some_file.svg") id "button" post "2".

SVG move ("some_file.svg") id "button1" to 10.0, 10.0.
SVG move ("some_file.svg") id "button2" to 30.0, 10.0.
```

Save as SVG

During simulation, the SVG visualizer can be used to save the currently visible image as an SVG image file. This way, the actual XML representation of the SVG image as it is displayed at that time, can be inspected in an XML editor or text editor. This may give insight in the actual effect of the CIF/SVG declarations applied until that moment.

Stack traces

If the application of a CIF/SVG declaration results in a runtime error, a stack trace is printed to the console. This stack trace makes it possible to track the exact origin of the error. Here is an example of such a stack trace for an output mapping:

```
ERROR: Simulation resulted in a runtime error.
CAUSE: Evaluation of the SVG output mapping ("some_file.svg") for the text of the SVG
element with id "txt1" failed for state: time=0.0, x=-1.5265566588595902e-16.
CAUSE: Evaluation of SVG output mapping value "sqrt(x)" failed.
CAUSE: Invalid operation: sqrt(-1.5265566588595902e-16).
```

Here, we see that simulation resulted in a runtime error. The cause is that the SVG output mapping for the text of the SVG element with id `txt1` failed. This points to the exact CIF/SVG mapping that failed, and includes the file name of the SVG image file, as the same id may exist in multiple images. It also includes the state at which the error occurred. The stack trace further indicates that evaluation of the value expression failed, and the actual failure was the evaluation of the `sqrt` standard library function, with a negative argument.

Console debugging for CIF/SVG real-time timing

Console debugging output for CIF/SVG real-time timing can be enabled or disabled by using the *Debug output* option (*Output* category). It is disabled by default. If enabled, it produces debug output to the console, whenever a new frame is displayed for an SVG visualizer. Here is an example of what such output looks like:

```
SVG#1: frame#1: no sleep for first frame...
SVG#1: frame#2: slept for (ms): 26, real-time time (ms): 50.6353, model time (ms):
50.0, total behind on real-time (ms): 0.6353000000000009, delta behind on real-time
(ms): 0.6353000000000009
SVG#1: frame#3: slept for (ms): 49, real-time time (ms): 104.5802, model time (ms):
100.0, total behind on real-time (ms): 4.580200000000005, delta behind on real-time
(ms): 3.944900000000004
```

The first frame is always immediately displayed. Subsequent frames may be delayed by a sleep to ensure they are displayed at exactly the right time. Each line for a non-first frame contains the following information:

- The number of the SVG image, to distinguish the output for different SVG images in case the CIF model has multiple SVG images associated to it.
- The frame number.
- The sleep time, as the number of milliseconds that the simulator slept before displaying this frame, after displaying the previous frame.
- The time that has passed since the simulation started.
- The model time for which the frame was rendered.
- How much behind the simulator is in real-time time, in total.
- How much behind the simulator is in real-time time, for this frame with respect to the previous frame.
- Optionally, if the sleep time is non-positive, the line explicitly indicates that the simulator did not sleep between displaying this frame and the previous frame.

This information can be used to see whether the simulator can keep up with calculating new states, rendering frames and displaying them in time before the next one needs to be displayed. As long as the sleep times are non-negative, the simulator can keep up. If the sleep times are negative, and thus the sleep is skipped, the simulator could not keep up. Note that the simulator does not try to catch up later on; it only tries to display each frame after the right amount of time has passed since the previous frame was displayed. In case the simulator cannot keep up, you may want to reduce the frame rate.

SVG viewer and visualizer

The Eclipse ESCET IDE comes with an SVG viewer. SVG image files can be opened for viewing. This same viewer is used as a basis for the SVG visualizer used by the CIF simulator.

SVG viewer

Opening SVG files in the ESCET IDE results in them being opened by the SVG viewer. The SVG viewer can be used to preview SVG images from within the ESCET IDE.

If the SVG viewer can not open an SVG image, for instance because it contains unsupported features such as text areas, the SVG viewer will indicate this. Therefore, to some extent, you can open an SVG image in the ESCET IDE to verify that it can be used for SVG visualization. Note however that not all unsupported features are reported this way, such as unsupported style properties in `style` attributes.

Refresh

The SVG viewer does *not* refresh its display if the image file is changed. To refresh the image, close the image, and open it again.

Save as

The SVG viewer and SVG visualizer can export the currently visible image to several different image formats. To export the image, first make sure that the SVG viewer or SVG visualizer has the focus. Then select **File › Save As...** to open the *Save SVG as* dialog. Alternatively, right click the SVG image itself, and choose **Save As...** from the popup menu, to open the *Save SVG as* dialog. In the dialog, specify the file name of the exported image. Click **[OK]** to confirm and to actually export the image.

The following raster image formats are supported:

- Portable Network Graphics (`*.png`)
- JPEG (`*.jpg`)
- Graphics Interchange Format (`*.gif`)

The following vector image formats are supported:

- Scalable Vector Graphics (`*.svg`)

The image format that is used to export the image, is derived from the file extension that is used. For each of the supported file formats, the allowed file names (with file extensions) are indicated above (between parentheses).

After the image is exported, the workspace is refreshed to show that new image file, if the image was saved in a project that is visible in the *Project Explorer* tab or *Package Explorer* tab.

Trajectory data output

The trajectory data (values of the variables as time progresses), calculated by the ode solver, can be written to a file, using the trajectory data output component.

Enable the *Trajectory data* option (*Output: Trajectory data* category) to enable writing the trajectory data to a file.

By default, the file to which to write the trajectory data, is the name of the input specification file, with `.trajdata` added to it. Using the *Trajectory data file* option (*Output: Trajectory data* category), a different file can be specified. The value of the option is the absolute or relative local file system path to use for the trajectory data file.

Filtering

Variable `time` is always included in the trajectory data file. The following additional variables (officially 'state objects') are included by default:

- The state variables, which includes the discrete variables, input variables and the continuous variables.
- The derivatives of the continuous variables. This does not include the derivative of variable `time`.
- The algebraic variables.

Only variables of type `bool`, `int` (with or without ranges), or `real` are included.

Using the *Trajectory data filters* option (*Output: Trajectory data* category), the state objects can be filtered. The *Trajectory data filters* option only has effect if trajectory data output is enabled, using the *Trajectory data* option.

As value for the option, comma separated *filters* should be supplied. Each filter specifies one or more state objects. The absolute names of the objects are used. That is, for an automaton `a`, with a variable `x`, the absolute name of the variable object is `a.x`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in the names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. If a filter doesn't match any of the state objects of the CIF model (after filtering them for supported types etc), a warning is printed to the console. A warning is also printed if the entire state is filtered out.

By default, filters include matching state objects. Filters may however be preceded by a `-` character, turning them into exclusion filters, which exclude matching states objects rather than including them. Filters are processed in the order they are specified, allowing for alternating additions and removals. If a filter does not result in the addition/removal of any state objects to/from the filter result, a warning is printed to the console.

As an example, option value `a.*,-a.b*,a.bc*` consists of three filters: `a.*`, `-a.b*`, and `a.bc*`. The first filter indicates that state objects whose absolute names start with `a.` are to be included. The second filter indicates that from those matching state objects, the state objects whose absolute names start with `a.b*` are to be excluded. To that result, the third filter adds those state objects whose absolute

names start with `a.bc*`. For instance, if a specification contains state objects `time`, `a.a`, `a.b` `a.bb`, `a.bc`, `a.b.c`, `a.bc`, `a.bcc`, and `a.bcd`, the result of the three filters is that the following state objects are displayed: `a.a`, `a.bc`, `a.bc`, `a.bcc`, and `a.bcd`.

The default option value (filter) is `*`.

Output

The variables are sorted alphabetically, based on their absolute names. The only exception is variable `time`, which is always listed first.

For variables with a boolean type, value `false` is represented by `0`, and value `true` is represented by `1`.

For each time transition, the trajectories of the variables are printed to the trajectory data file. The first line is a comment, and contains the names of the variables. Each of the subsequent lines contain the values of the variables for a specific value of variable `time`. The exact times for which the values are printed to the trajectory data file, is determined by the ODE solver integrator, and can be influenced using the integration options, as well as the fixed output step size option.

An example of a trajectory data file, for a single time transition:

```
# time controller.n tank.V tank.V' tank.Qi tank.Qo
0.0 0 10.0 -3.1622776601683795 0.0 3.1622776601683795
0.08763429266802605 0 9.7247959763441 -3.1184605138343664 0.0 3.1184605138343664
0.587634292668026 0 8.228065719426915 -2.868460513834366 0.0 2.868460513834366
1.087634292668026 0 6.8563354625097315 -2.618460513834366 0.0 2.618460513834366
1.587634292668026 0 5.609605205592543 -2.3684605138343646 0.0 2.3684605138343646
2.0876342926680262 0 4.487874948675351 -2.1184605138343624 0.0 2.1184605138343624
2.5876342926680262 0 3.491144691758149 -1.8684605138343569 0.0 1.8684605138343569
3.0876342926680262 0 2.619414434840913 -1.618460513834339 0.0 1.618460513834339
3.496128195593086 0 1.9999999999999958 -1.4142135623730936 0.0 1.4142135623730936
```

Prettifying

To make the output in the trajectory data file easier to read, the output can be *prettified*. If prettifying is enabled, then after the simulation ends, the trajectory data file is read, prettified, and rewritten. A fixed width is then used for each column (each variable), with a fixed column separation. After prettification, the above trajectory data file looks like this:

```
# time                controller.n   tank.V              tank.V'
tank.Qi   tank.Qo
0.0                   0              10.0                -3.1622776601683795   0.0
3.1622776601683795
0.08763429266802605   0              9.7247959763441     -3.1184605138343664   0.0
```

```
3.1184605138343664
0.587634292668026     0             8.228065719426915    -2.868460513834366    0.0
2.868460513834366
1.087634292668026     0             6.8563354625097315   -2.618460513834366    0.0
2.618460513834366
1.587634292668026     0             5.609605205592543    -2.3684605138343646   0.0
2.3684605138343646
2.0876342926680262    0             4.487874948675351    -2.1184605138343624   0.0
2.1184605138343624
2.5876342926680262    0             3.491144691758149    -1.8684605138343569   0.0
1.8684605138343569
3.0876342926680262    0             2.619414434840913    -1.618460513834339    0.0
1.618460513834339
3.496128195593086     0             1.9999999999999958   -1.4142135623730936   0.0
1.4142135623730936
```

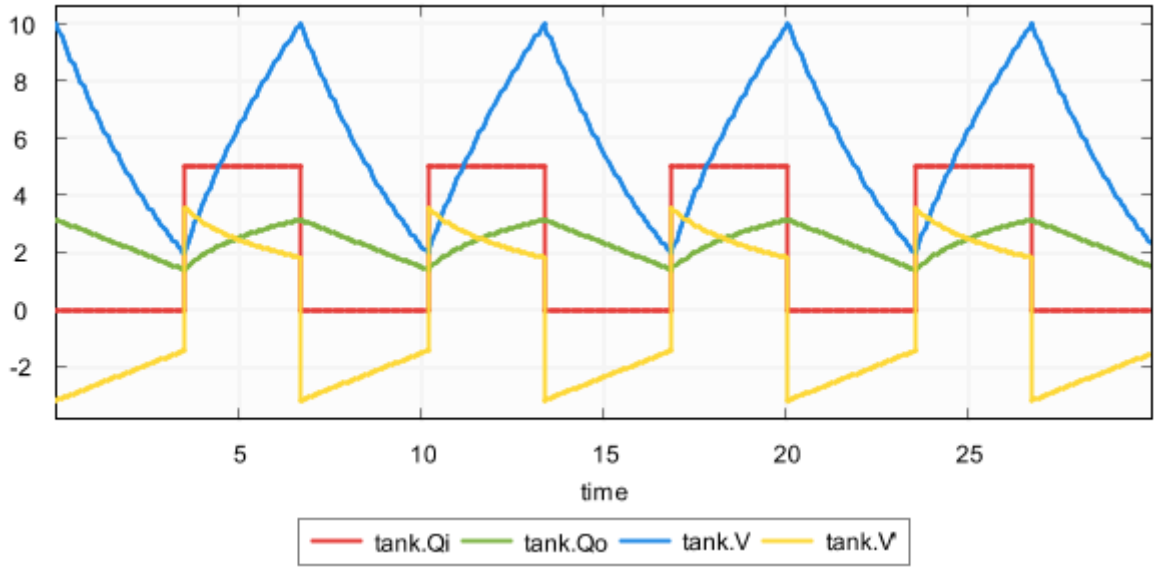
By default, the trajectory data output is prettified using a column separation of three spaces. This can be changed using the *Trajectory data separation amount* option (*Output: Trajectory data* category). Using this option, prettification can be disabled, or a different number of spaces can be used between the columns.

Undo/reset

It is possible to go back in time, by undoing one or more transitions, or by resetting the simulation. However, once a time transition is taken, trajectory data is written to the output file. The simulator does not remove data for time points that are then suddenly in the future. If simulation continues after the undo or reset, new data may be written at the end of the trajectory, for the same time points that were already present earlier in the data file. As such, the data contained in the trajectory data file, may no longer be useful after an undo or reset, if simulation continues afterwards.

Plot visualizer

For models with variables, especially continuous ones, it may be useful to be able to observe how their values change, as time progresses during simulation. The *plot visualizer* can be used for that exact purpose. The plot visualizer can create graphical plots, where the x-axis represents the values of variable `time`, and the y-axis represents values of the variables being plotted. Here is an example screenshot of the visualizer:



Plot visualization is disabled by default. It can be enabled using the *Plot visualization* option (*Output: Plot visualization* category).

Variables

The plot visualizer can plot the values of the following variables (officially 'state objects'):

- The state variables, which includes the discrete variables, input variables and the continuous variables.
- The derivatives of the continuous variables. This does not include the derivative of variable `time`.
- The algebraic variables.

Only variables of type `bool`, `int` (with or without ranges), or `real` can be plotted. For variables with a boolean type, value `false` is represented by `0`, and value `true` is represented by `1`.

Filtering

By default, if plot visualization is enabled, all variables (officially 'state objects') that can be plotted are plotted. The one exception is variable `time`, which is excluded by default, as it already represents the x-axis. That is, variables are plotted against time. Using the *Plot visualization filters* option (*Output: Plot visualization* category), the state objects can be filtered. The *Plot visualization filters* option only has effect if plot visualization is enabled, using the *Plot visualization* option.

As value for the option, comma separated *filters* should be supplied. Each filter specifies one or more state objects. The absolute names of the objects are used. That is, for an automaton `a`, with a variable `x`, the absolute name of the variable object is `a.x`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in the names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. If a filter doesn't match any of the state objects of the

CIF model, a warning is printed to the console. A warning is also printed if the entire state is filtered out.

By default, filters include matching state objects. Filters may however be preceded by a `-` character, turning them into exclusion filters, which exclude matching states objects rather than including them. Filters are processed in the order they are specified, allowing for alternating additions and removals. If a filter does not result in the addition/removal of any state objects to/from the filter result, a warning is printed to the console.

As an example, option value `a.*,-a.b*,a.bc*` consists of three filters: `a.*`, `-a.b*`, and `a.bc*`. The first filter indicates that state objects whose absolute names start with `a.` are to be included. The second filter indicates that from those matching state objects, the state objects whose absolute names start with `a.b*` are to be excluded. To that result, the third filter adds those state objects whose absolute names start with `a.bc*`. For instance, if a specification contains state objects `time`, `a.a`, `a.b` `a.bb`, `a.bc`, `a.b.c`, `a.bc`, `a.bcc`, and `a.bcd`, the result of the three filters is that the following state objects are displayed: `a.a`, `a.bc`, `a.bc`, `a.bcc`, and `a.bcd`.

The default option value (filter) is `*,-time`.

Multiple plot visualizers

By default, only one visualizer is shown. However, using the *Plot visualization filters* option (*Output: Plot visualization* category), it is possible to specify that multiple visualizers should be used. The option allows for filtering of the state objects, as described above. However, such filters can be separated by semicolons, to specify the filters per visualizer.

As an example, consider option value `time,a.x;b.y`. This results in two plot visualizers. The first one displays state objects `time` and `a.x`, while the second one displays state object `b.y`.

Plot visualization modes

There are two plot visualization modes:

- Live plotting. In this mode, the plots are shown at the start of the simulation, and are continuously updated as new data becomes available during simulation.
- Postponed plotting. In this mode, the plots are shown after the simulation has ended.

Which plot visualization mode to use, can be configured using the *Plot visualization mode* option (*Output: Plot visualization* category). Using that option it is possible to explicitly choose one of the modes. By default, an automatic mode is used, which chooses between live and postponed mode, as follows:

Input mode vs real-time simulation	Real-time enabled	Real-time disabled
Interactive console input mode (pure interactive)	live	live
Interactive console input mode (semi-automatic)	live	postponed
Interactive GUI input mode (pure interactive)	live	live
Interactive GUI input mode (semi-automatic)	live	postponed
Automatic input mode	live	postponed
Trace input mode	live	postponed
SVG input mode	live	postponed

When doing a non real-time simulation, using a non-interactive input mode leads to as fast as possible simulation, where a lot of data points are calculated in a short amount of time. If live plotting mode is then enabled, this floods the visualizer with so much data that it can't keep up. The effect is a non-responsive user interface. While the automatic default can thus be overridden using the option, it is generally not recommended.

Plot visualization range

For simulations that span longer periods of (model) time, there may be too much data to display, for the limited width of the time axis (x-axis). To keep the plot useful, the range of time values to display on the x-axis can be configured using the *Plot visualization range* option (*Output: Plot visualization* category).

By default, only the `50` most recent time units are displayed. That is, if the current model time is `120`, then the plot only shows the values of the variables from time `70` to time `120`. Using the option, the length of the range can configured to any other positive value. The option can also be used to set the range to infinite, to display the values of the variables during the entire simulation, from time `0` to the current model time.

For long running simulations, with lots variables, using an infinite range can lead to large numbers of data points, which can have a significant effect on performance. This applies especially for live plotting mode.

Data points

This section describes which data points are visualized. For most users, this will be of little interest, as it essentially works as you would expect.

The plot visualizer adds data points (a `time` value and a value of a variable at that time) for all

variables, for the following states:

- The initial state of the simulation.
- The start state of every time transition.
- Intermediate states of time transitions, only if real-time simulation is enabled.
- Trajectory data points of time transitions, only if real-time simulation is disabled.
- The end state of every time transition.

For real-time simulation, the amount of model time between two intermediate states is the model time delta, and can be influenced using the frame rate and simulation speed.

For non real-time simulation, the trajectory data points are determined by the ODE solver integrator, and can be influenced using the integration options, as well as the fixed output step size option.

Relation to trajectory data output

The plot visualizer can be used for simple plotting. It can be customized a bit through options, as described above. However, the level of customization is somewhat limited. For instance, the appearance can not be customized. This is intentional.

If further customization is required, use the trajectory data output component instead. It allows saving the data to a file, for further processing with external tools, such as R.

Such post processing is then performed after the simulation has ended. A benefit of the plot visualizer, is that it allows live plotting mode, without the need of post processing, and which can be enabled with little effort. The plot visualizer is meant to be used only to get a basic understanding of how values of variables change as time progresses.

Saving a plot image

The plot visualizer can export the currently visible plot to several different image formats. To export the image, first make sure that the plot visualizer has the focus. Then select **File › Save As...** to open the *Save plot as* dialog. Alternatively, right click the plot itself, and choose **Save As...** from the popup menu, to open the *Save plot as* dialog. In the dialog, specify the file name of the exported image. Click **[OK]** to confirm and continue.

A second dialog appears, in which the size of the exported image can be specified. By default, the current size of the visualizer is used. A custom size can be entered. The width and height need to specified in pixels, separated by a `x` character. For instance, for a width of `640` pixels and a height of `480` pixels, enter `640x480` into the dialog. Click **[OK]** to confirm and to actually export the image.

The following raster image formats are supported:

- Portable Network Graphics (`*.png`)

- JPEG (`*.jpg`)
- Graphics Interchange Format (`*.gif`)

The image format that is used to export the image, is derived from the file extension that is used. For each of the supported file formats, the allowed file names (with file extensions) are indicated above (between parentheses).

After the image is exported, the workspace is refreshed to show that new image file, if the image was saved in a project that is visible in the *Project Explorer* tab or *Package Explorer* tab.

In order for the export to succeed, data points must be available for at least two time values.

Undo/reset

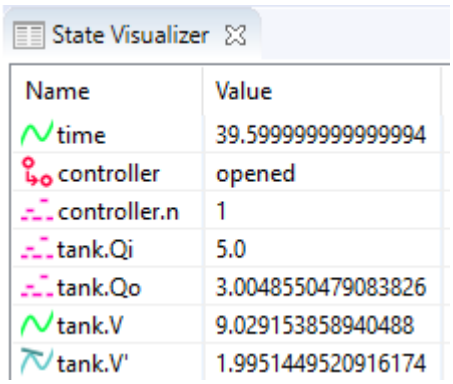
It is possible to go back in time, by undoing one or more transitions, or by resetting the simulation.

If one or more transitions are undone, data for all time points that are then suddenly in the future, are removed. More precisely, all data for time points added for time transitions that have been undone, are removed.

If the simulation is reset, the entire plot is cleared.

State visualizer

The normal console output, which may include the state of the specification, is printed to the console. The entire state is then printed as a single line of text, which can be hard to read as the specification grows, and the state contains more and more variables. The *state visualizer* can be used as an alternative. It shows the same information, but in a table in a separate window. Here is an example screenshot of the visualizer:



Unlike the SVG visualizer, which requires that the user makes an image of the system and couples the image to the specification, the state visualizer can be used directly, without any additional effort. All that is required to use it, is to enable it. The state visualizer is disabled by default, and can be enabled using the *State visualization* option (*Output: State visualization* category).

Contents

The state visualizer shows all the discrete, input, continuous, and algebraic variables of the specification, as well as their values. It also shows the derivatives of the continuous variables, and their values. This includes variable `time`, but not its derivative. It also shows the automata, with their current locations. The variables and automata are listed in alphabetical order, except for variable `time`, which is always shown at the top.

Icons and legend

The icons on the left indicate the kind of object that is shown in a particular row of the table:

-  An automaton, with its current location.
-  A discrete variable, with its current value.
-  An input variable, with its current value.
-  A continuous variable (includes variable `time`), with its current value.
-  A derivative of a continuous variable (excludes variable `time`), with its current value.
-  An algebraic variable, with its current value.

The values are updated for the initial state, states reached after transitions (event transitions and time transitions), as well as for frames.

Filtering

By default, all state objects are displayed by the state visualizer. Using the *State visualization filters* option (*Output: State visualization* category), the state objects can be filtered. The *State visualization filters* option only has effect if state visualization is enabled, using the *State visualization* option.

As value for the option, comma separated *filters* should be supplied. Each filter specifies one or more state objects. The absolute names of the objects are used. That is, for an automaton `a`, with a variable `x`, the absolute name of the variable object is `a.x`. If CIF textual syntax keyword are used as names for events (such as `plant`), then they must be escaped in `.cif` files (`$plant`). For filters however, all escape characters (`$`) in the names are ignored. The `*` character can be used as wildcard, to indicate zero or more characters. If a filter doesn't match any of the state objects of the CIF model, a warning is printed to the console. A warning is also printed if the entire state is filtered out.

By default, filters include matching state objects. Filters may however be preceded by a `-` character, turning them into exclusion filters, which exclude matching states objects rather than including them. Filters are processed in the order they are specified, allowing for alternating additions and removals. If a filter does not result in the addition/removal of any state objects to/from the filter result, a warning is printed to the console.

As an example, option value `a.*,-a.b*,a.bc*` consists of three filters: `a.*`, `-a.b*`, and `a.bc*`. The first

filter indicates that state objects whose absolute names start with `a.` are to be included. The second filter indicates that from those matching state objects, the state objects whose absolute names start with `a.b*` are to be excluded. To that result, the third filter adds those state objects whose absolute names start with `a.bc*`. For instance, if a specification contains state objects `time`, `a.a`, `a.b` `a.bb`, `a.bc`, `a.b.c`, `a.bc`, `a.bcc`, and `a.bcd`, the result of the three filters is that the following state objects are displayed: `a.a`, `a.bc`, `a.bc`, `a.bcc`, and `a.bcd`.

The default option value (filter) is `*`.

Multiple state visualizers

By default, only one visualizer is shown. However, using the *State visualization filters* option (*Output: State visualization* category), it is possible to specify that multiple visualizers should be used. The option allows for filtering of the state objects, as described above. However, such filters can be separated by semicolons, to specify the filters per visualizer.

As an example, consider option value `time,a.x;b.y`. This results in two state visualizers. The first one displays state objects `time` and `a.x`, while the second one displays state object `b.y`.

Undo/reset

By undoing one or more transitions, or by resetting the simulation, the state may change. The state visualizer is automatically updated to reflect such changes.

Real-time simulation

The CIF simulator allows for a form of real-time simulation. By enabling real-time simulation, the simulator interprets the model time (the value of variable `time`) in seconds. The simulator attempts to ensure that the actual simulation time is equal to the model time. This feature is very useful for visualizers, which then visualize their output at a steady rate. Without real-time simulation, the rate at which such visualizers would visualize their output, would primarily be determined by the amount of time it takes for the simulator to compute the output, leading to abrupt slow downs and sudden speed increases.

Frames

If real-time simulation is disabled, visualizations are only updated for the states that are reached. That is, the visualizations are updated for the initial state, and each new state that is reached after taking a transition. The visual representations of the visualizations resulting from these states are called *frames*.

As time progresses, the values of continuous variables (and algebraic variables for which the value is directly or indirectly derived from a continuous variable, etc) change. If the visualization is only updated before and after time transitions, as is done when real-time simulation is disabled, this

leads to 'jumps' in the visualization. For real-time simulation, the simulator calculates states that are evenly spaced over time. This ensures that we get regular updates of the visualizations, as time progresses, and the visualizations thus become 'smooth' or 'fluent'. If enough states are visualized per second, a visualization essentially becomes a *movie*.

The states that result from transitions are always calculated. For real-time simulation, the simulator calculates additional states for the purpose of visualization, as mentioned above. These additional states are called *intermediate states*, and the corresponding visual representations of the visualizations to which they lead are called *intermediate frames*.

For real-time simulation, the simulator thus uses intermediate frames that are evenly spaced over time, for smooth visualizations. A consequence of this, is that state changes between those intermediate frames are lost. If the effect is still visible in the next intermediate state, this is not a problem. If however the effect is temporary, and does not last until the next intermediate state, the effect is thus not visualized.

For real-time simulation combined with an interactive input mode, the simulator uses both the state frames and the intermediate frames. This ensures that at the moment an interactive choice has to be made about which transition to take, the visualization reflects the current state of the system. Semi-automatic mode is not taken into account.

Frame rate

Using the *Frame rate* option (*Output* category), it is possible to specify how often per second of real time, visualizations should be updated. A higher frame rate leads to more updates per second, and thus to a more fluent visualization. However, since a higher frame rate also leads to more frequent updates, the amount of CPU power it takes to perform these updates increases as well. If the computer can not update the visualization in time for the next update to take place, the visualization may start to 'stutter'.

The frame rate will thus always be a compromise between fluency and performance. Choosing an appropriate frame rate depends not only on the complexity of the specification being simulated and the visualization that is being shown, but also on the computer on which the simulation is run. Furthermore, it may depend on the other tasks the computer is performing while the simulation is being run.

Frame rates are always positive, and may be fractional. For instance, a frame rate of `7.5` is valid, and will update the visualization 7.5 times per second.

A low frame rate (typically less than 10 frames per second) can have a devastating effect on the responsiveness of SVG input. Similarly, it can hinder termination of the simulation, as the red *Terminate* button has no effect between intermediate frames. That is, the effect of clicking the button is postponed until the next frame, or until the time transition ends.

To analyze the real-time timing of frames visualized by the SVG visualizer, and see whether the simulator can keep up with your chosen frame rate, you can enable the CIF/SVG real-time timing debug output.

Simulation speed

If real-time simulation is enabled, the simulator by default interprets the model time (the value of variable `time`) in seconds. However, this can be changed using the *Simulation speed* option (*Output* category). This option specifies a scaling factor. A scaling factor of `1.0` interprets model time as seconds, `2.0` is twice as fast (1 unit of model time takes 0.5 seconds), etc. The simulation speed must always be positive, and may be fractional. For instance, a simulation speed of `0.5` is valid, and will interpret model time as twice as slow (1 unit of model time takes 2 seconds).

Enabling real-time simulation

By default, real-time simulation is disabled, and the simulator will try to simulate as fast as possible (infinite speed, taking time transitions takes no time). By setting a frame rate and/or a simulation speed, real-time simulation can be enabled. If only one of the options is specified, the other gets a default value. The default frame rate for real-time simulation is `20.0` frames per second. The default simulation speed is `1.0`.

Enabling real-time simulation requires a real-time output component (visualizer). Currently, the following real-time output components are available:

- SVG visualizer

If real-time simulation is enabled without using a real-time output component, simulation can't be started. Closing the visualizer of a real-time simulation during the simulation, if it is the last remaining real-time output component, leads to the simulation being stopped.

All real-time output components can also be used when real-time simulation is disabled, but obviously without any real-time effects.

Model time delta

The simulation speed does not influence the frame rate. That is, both options can be specified orthogonally. The two options together however, do influence the model time delta, the amount of model time between two consecutive updates of the visualizations, i.e. the amount of model time between two consecutive intermediate frames. The model time delta is calculated using the following formula (where `r` is the frame rate, and `s` is the simulation speed): 1 / (`r` / `s`). Doubling the frame rate results in half the model time delta, while doubling the simulation speed results in double the model time delta.

If the model time delta becomes to big, this may lead to 'jumps' in the visualization. In such cases, increasing the frame rate, or reducing the simulation speed causes a smaller model time delta, leading to a more fluent visualization.

The model time delta (and thus the frame rate and simulation speed) also affects the ODE solver fixed output step size.

Test mode

Test mode disables all real-time delays of the real-time simulation mode, but still updates the visualizations for the intermediate frames. With test mode enabled, it is also allowed to perform a real-time simulation without real-time output component.

Simulation advanced topics

Simulator options

The CIF simulator can be configured via various options. The available options are listed below, per category.

General

- *Help*: Prints the help text of the application to the console.
- Output mode
- *License*: Prints the license text of the application to the console.
- *Developer mode*: Option to control whether exceptional situations return limited and user-friendly information (the default), or extended developer oriented information. This option is primarily intended for the developers of the CIF simulator, and should not be used by end users.

Simulator

- *CIF specification*: The absolute or relative local file system path to the CIF specification (`.cif` or `.cifx` file) to simulate. May also be a compiled code file (`.cifcode` file).
- CIF specification initialization
- Simulation end time
- Maximum delay
- Initial seed for stochastic distributions
- Execution mode
- Complete mode
- Profiling
- Ask to confirm simulator termination
- Maximum time point tolerance
- Load debug code

Compiler

- Compile only

- Compiled code file
- Java compiler

Input

- Input mode
- Automatic mode choice algorithm
- Automatic mode time transition duration
- Trace input file
- SVG input outline style
- SVG input outline filters
- Interactive mode automatic transition choice
- Non-urgent events
- History
- History size

Output

- Normal output
- Normal output state filters
- Debug output
- Frame rate
- Simulation speed
- Test mode

Output: SVG visualization

- SVG visualization

Output: Trajectory data

- Trajectory data
- Trajectory data file
- Trajectory data filters
- Trajectory data separation amount

Output: Plot visualization

- Plot visualization
- Plot visualization mode
- Plot visualization filters
- Plot visualization range

Output: State visualization

- State visualization
- State visualization filters

ODE solver

- ODE solver fixed output step size

ODE solver: ODE integrator

- ODE solver integrator algorithm
- ODE solver integrator absolute tolerance
- ODE solver integrator relative tolerance
- ODE solver integrator minimum step size
- ODE solver integrator maximum step size
- ODE solver integrator number of steps

ODE solver: ODE root finder

- ODE solver root finder maximum check interval
- ODE solver root finding algorithm
- ODE solver root finder absolute tolerance
- ODE solver root finder relative tolerance
- ODE solver root finder maximum iterations

Simulation performance

General performance information about Java or Eclipse typically applies to the CIF simulator as well. This page provides additional information specific to the CIF simulator.

Closing the simulator

Closing the simulator ensures that all its resources are freed, and become available for other applications. The CIF simulator however, may ask you to press `Enter` to confirm termination of the simulator. If asked, as long as you haven't pressed `Enter` yet, the status of the console will still show it's running, and the application can't release its resources.

Slow starting of the simulator

If simulation is slow to start, you can try a different Java compiler. You can also try to compile the model once, reducing the start time of the simulator for repeated simulations.

Slow termination of the simulator

If termination of the simulator is slow, this may be related to the trajectory data output component. If its prettifying option is enabled, it will read the trajectory data file after the simulation

terminates, and write the whole file again in a prettier from. This may take some time, especially on slow remote/network file systems or storage devices. Disabling the option or the trajectory data output may solve this problem, as may switching to a faster file system or storage device.

Value simplification

By applying the Simplify values CIF to CIF transformation before simulation, you may be able to simplify the specification, and thus improve the performance of both starting up the simulator, as well as the actual simulation.

Simulation options

Various simulation options can be tweaked to increase the simulation performance:

- Console output

  Reducing the amount of console output significantly improves simulation performance.

- Output components

  By disabling certain output components, the simulator needs to do less work, and this may improve the performance of the simulation.

- Real-time simulation

  The performance and perceived 'smoothness' or 'fluency' of visualizations can be influenced via the frame rate and simulation speed.

- Maximum delay

  By decreasing the maximum allowed length of a single time transition, shorter time transitions are calculated, which takes less time. After the shorter time delay, the simulator will calculate the remainder of the time transition. Essentially, the time transitions are cut into parts, which are calculated separately, over time. As such, the calculation time is spread out over time as well. This can make SVG visualizations of the SVG visualizer feel more fluent. However, each time transition calculation has a certain amount of overhead, so reducing the maximum delay too much is not a good idea.

- Complete mode

  By disabling the complete mode, the simulator has to perform less work to calculate the possible transitions, improving the simulation performance.

- Solver

  The ODE solver options can be used to make the ODE solver faster, usually at the expense of accuracy. For instance, increasing the various tolerances reduces the number of calculations needed by the ODE solver, but makes the calculated trajectories less accurate. Switching algorithms may also improve the performance, as may increasing the maximum check interval.

ODE solver

During time transitions, the values of the continuous variables and their derivatives change. These changes are calculated using an ODE integrator. As soon as a guard for an event becomes enabled, time should no longer progress. To calculate whether (and when) guards change value, a root finding algorithm is used. Software to solve ODEs, through integration and root finding, are often called ODE solvers. The CIF simulator uses the The Apache Commons Mathematics Library. This library contains ODE solvers, with several different integrators and root finding algorithms.

Debug information about what the ODE solver calculates, and for what time points, can be enabled using the *Debug output* option. See the Debug console output page for more information.

Basic topics

- The basics of numeric integration
- The basics of numeric root finding
- Problems with root finding (and missed guards/events)
- Fixed output step size

Advanced topics

- Simulation end time and maximum delay
- ODE solver integrator options
- ODE solver root finder options

The basics of numeric integration

During time transitions, the values of the continuous variables and their derivatives change. The derivatives have explicit equations, the continuous variables change according to the value of their derivatives. Using the equations for the derivatives as a system of ordinary differential equations (ODEs), together with the initial values of the continuous variables as the *initial conditions*, this essentially comes down to solving an initial value problem (IVP).

Such problems can be solved through integration. For some problems it is possible to do this symbolically. For more complex systems of ODEs however, numerical methods are used. The CIF simulator uses The Apache Commons Mathematics Library, which contains several numerical integrators.

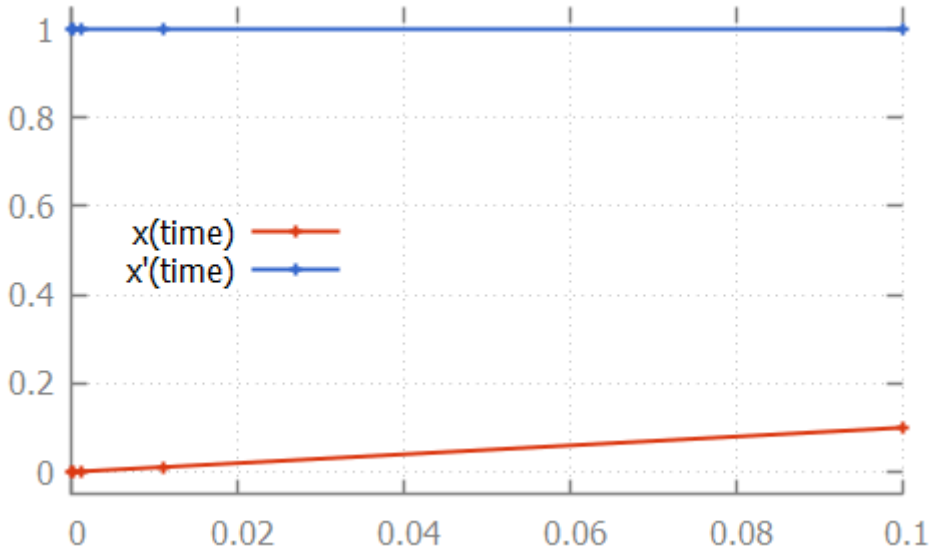
Linear ODE

Consider the following CIF specification, with a linear ODE:

```
cont x = 0.0;
equation x' = 1.0;
```

The solution to this IVP, is the values of continuous variable `x` and its derivative `x'`, as function of

variable `time`:



Here, the numerical integrator start with the initial value of continuous variable `x`, which is `0.0`. For the initial value of variable `time`, which is also `0.0`, it computes `x'`, which is `1.0`. It then slightly increases the value of variable `time` to say `1e-6`. Assuming a linear continuous variable `x`, the value of variable `x` at that time is `1e-6` as well.

The numerical integrator tries to predict the values of the derivative as time progresses. It gradually increases the value of variable `time`, predicting the value of the derivative at the next time point. If the next prediction closely matches the actual calculated value, the integrator moves on to the next time point. If the next prediction is not close enough to the actual calculated value of the derivative for that time point, more intermediate values are calculated. That is, the integrator tries to approximate the derivative as time progresses, while increasing the time between two consecutive time points. As long as the predictions match the actual calculated value of the derivative at the next time point, it keeps increasing the step size even further. If the predictions are not good enough (the difference with the actual calculated value is above a certain tolerance), more intermediate time points are investigated. The values calculated for those time points can then be used to come up with a better approximation, that better predicts the value of the derivative at future time points.

For the linear ODE given above, the trajectory data calculated by the integrator is:

```
# time                x                     x'
0.0                   0.0                   1.0
9.999999999999999e-5  1.0000000000000003e-4 1.0
0.0011                0.0011000000000000005 1.0
0.0111                0.011100000000000006  1.0
0.1                   0.10000000000000006   1.0
```

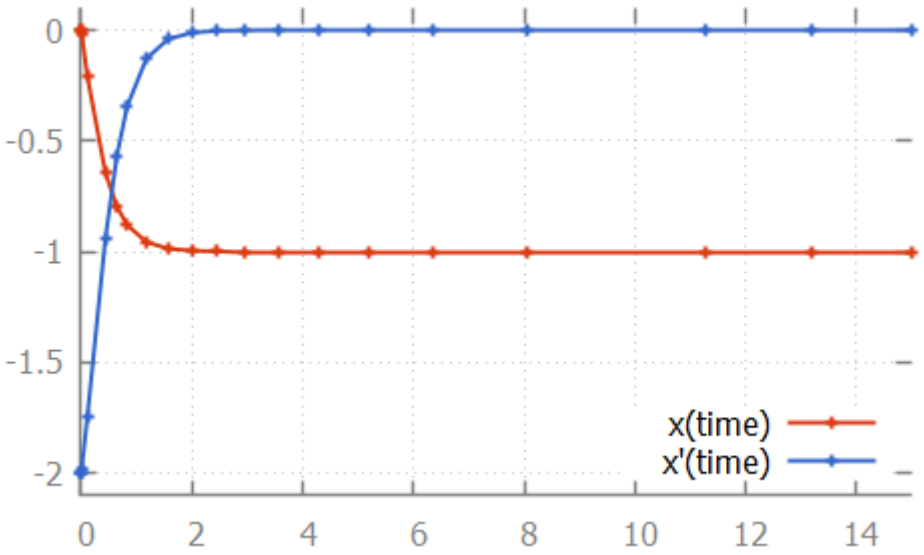
Note that the time points for which the values were calculated, are indicated in the figure above by small plus signs (`+`).

Nonlinear ODE

The approximations of the integrator don't need to be linear. Some integrators for instance support nonlinear differential equations. Consider the following CIF specification, with such a nonlinear ODE:

The solution calculated by the numerical integrator is:

```
cont x = 0.0;
equation x' = (x * x) - x - 2;
```



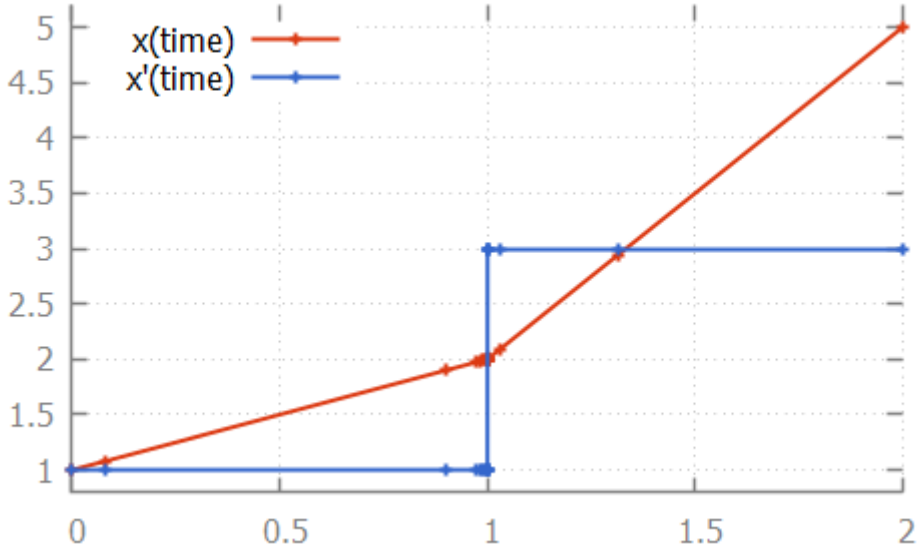
Here, we see how the step size is increased initially, as the linear approximation is good enough. As soon as we get to the bend, the step size is reduced to better approximate the actual values. After the bend, the step size is slowly increased again. The integrator internally uses a polynomial of a higher degree to approximate this differential equation.

Discontinuities

If the derivative has a discontinuity, such as in the following CIF specification:

```
cont x = 1.0;
equation x' = if x < 2: 1.0
              else      3.0
              end;
```

The integrator will try to figure out the time point at which the discontinuity occurs, by decreasing the step size as it nears the discontinuity. After the discontinuity, the step size is gradually increased, as integration continues:



The basics of numeric root finding

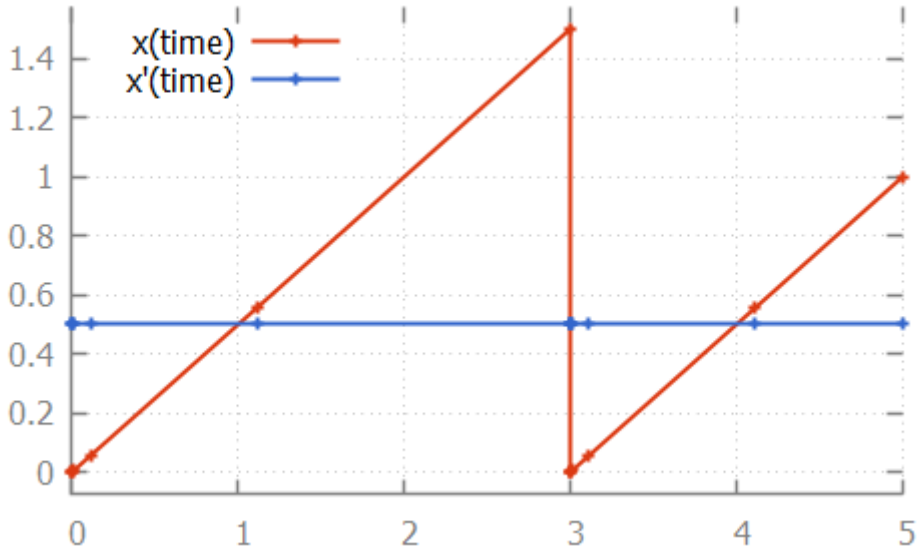
During time transitions, the values of the continuous variables and their derivatives change. If continuous variables or their derivatives are (directly or indirectly) used in guards of edges, changes in their values may result in the guard becoming enabled, as time progresses. To detect such changes during integration, a root finding algorithm can be used.

Consider the following CIF specification:

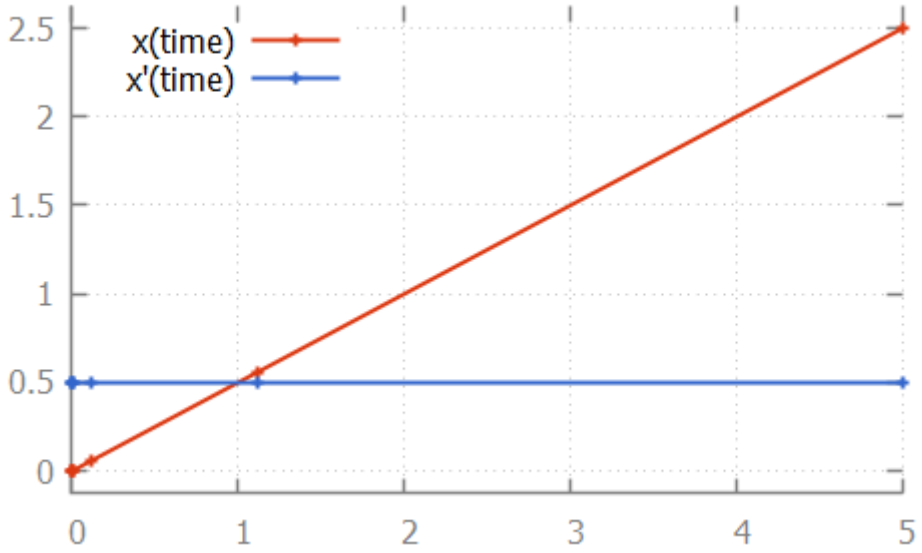
```
automaton p:
  cont x = 0.0;
  equation x' = 0.5;

  location:
    initial;
    edge when x >= 1.5 do x := 0.0;
end
```

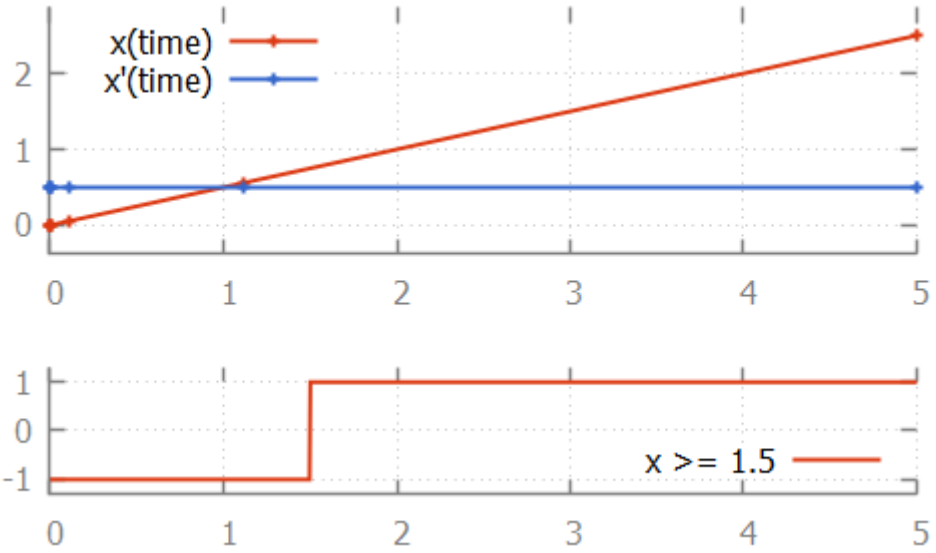
Here, the value of continuous variable `x` increases with `0.5` every time unit. Once the value of `1.5` is reached, the variable is reset to `0.0`. This process is repeated:



If we look at the data calculated by the numerical integrator, to solve the ODE problem:



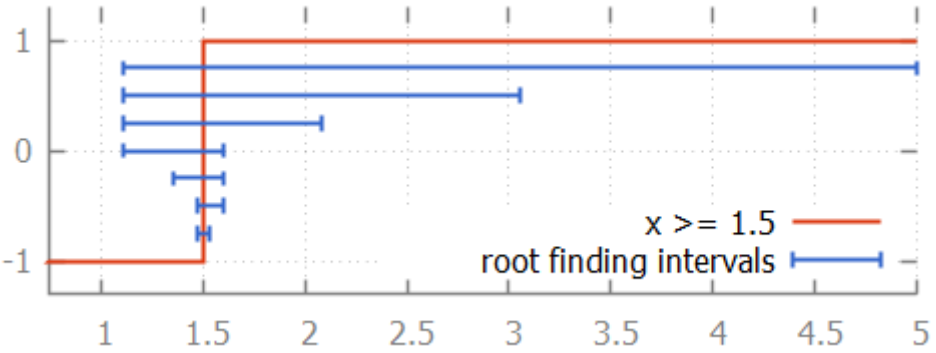
we see that values are calculated for time points `1.1111` and `5.0`. If we then also look at the value of the guard, where we use value `-1` for `false` and `+1` for `true`:



we see that at time `1.1111`, the value of the guard is `-1` (`false`). At time `5.0` it is `+1` (`true`). That is, the guard changed value between two time points calculated by the integrator. If this is the case, the ODE solver tries to calculate the exact time point at which the guard changes its value. That is, it calculates the exact time point at which the guard function crosses the time axis, and thus has a root.

One of the simplest root finding algorithms, is the bisection method. This algorithm starts with the two time points where the guard has opposite signs. This is the interval where the guard sign change occurs (it contains the root). The bisection method tries to reduce the size of the interval, by calculating the value of the guard in the middle of the interval. Depending on the sign of the value of the guard at this middle point, this middle point replaces either the lower bound or the upper bound of the interval. This is done in such a way that the values of the guard at the lower and upper bound of the interval have opposite signs, and the interval thus *brackets* the root.

For the example above, the root is calculated as follows:



We start with the interval `[1.1111 .. 5.0]`. The middle point is `3.0555` at which the guard holds, just like the upper bound (at `5.0`). Thus `5.0` is replaced by `3.0555`. The middle point of `1.1111` and `3.0555` is `1.5972`. Since the guard holds for time `1.5972`, upper bound `3.0555` is replaced by `1.5972`. The middle point of `1.1111` and `1.5972` is `1.3542`. The guard does not hold at time `1.3542`, so the lower bound of `1.1111` (at which the guard does not hold) is replaced by `1.3542`. This process continues until the interval is smaller than a certain tolerance value. Once we have that interval, we can choose a value from the interval as the computed root.

While the bisection method is relatively simple, the root finding algorithms used by the CIF simulator work using the same principles. However, they converge much faster. That is, they requires much less iterations of updating the bounds, to get to a satisfactory answer.

Problems with root finding

With numeric root finding, there is the risk of missing guard changes, which means there is the risk of missing event transitions.

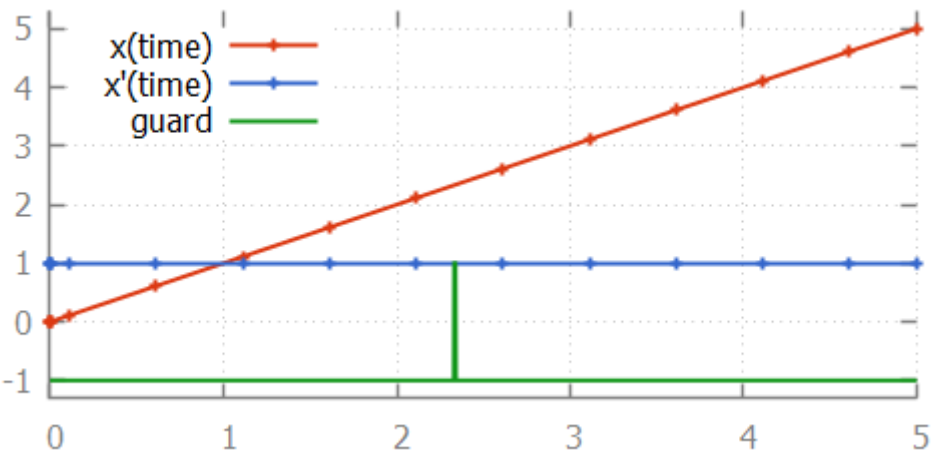
The problem

To illustrate the problem, consider the following CIF specification:

```
automaton p:
  cont x = 0.0;
  equation x' = 1.0;

  location:
    initial;
    edge when x >= 2.33 and x <= 2.34 do x := 0.0;
end
```

This specification, when simulated using the CIF simulator, using default settings, results in:



From the image, where the values calculated by the integrator are indicated using small plus signs (`+`), we can see that for times `2.1111` and `2.6111`, the integrator calculated points. Since the function is linear between those two time points, no additional intermediate time points were investigated. Since the guard becomes `true` after time point `2.1111` (at time point `2.33`) and also becomes `false` again before time point `2.61111` (at time point `2.34`), the integrator misses the guard change. That is, for every time point that the guard is evaluated, it is `false`, and the change in guard value is thus not detected. We miss the guard, and thus the event that resets variable `x` to `0.0`.

Solution 1: maximum check interval

The first solution is to use the *ODE solver root finder maximum check interval* option (*ODE solver: ODE root finder* category). This option can be used to set the maximum time interval between two consecutive checks of the sign of the guard. By default, this is `0.5` time units. If we set this value to `0.001`, then we ensure that at least once every `0.001` time units, the guard sign is checked. This means that between time points `2.1111` and `2.61111`, the guard will be checked approximately 500 times. This also means that between time points `2.33` and `2.34`, the guard will be checked approximately 10 times. Therefore, this ensures that we check a time point just before time `2.33`, and one just after it, which means we can't miss the guard change.

In the above example, `x'` is `1.0`. If however `x'` would be `2.0`, then the time interval where the guard is enabled reduces from `0.01` time units to `0.005` time units. Using a maximum check interval of `0.001` would then still suffice, as the guard would be checked approximately 5 times in that time interval. If however `x'` is `100`, then the time interval would become `0.0001`, which is smaller than the maximum check interval of `0.001`. That is, the derivatives of the values occurring in guards influence the value that should be chosen for the maximum delay interval.

In general, to choose an appropriate value for the maximum check interval, not only the ranges of values of the variables where the guard is enabled should be taken into account, but the steepness of their derivatives as well. That is, the value of the maximum check interval should be based on the time interval where the guard is enabled. Choosing half the duration of the smallest of such time intervals, for all guards, is generally a good choice.

Note that choosing a smaller interval results in more guard change checks, and thus negatively impacts the performance of the simulation. Therefore, choosing a maximum check interval that is unnecessarily small also unnecessarily reduces performance of the simulation.

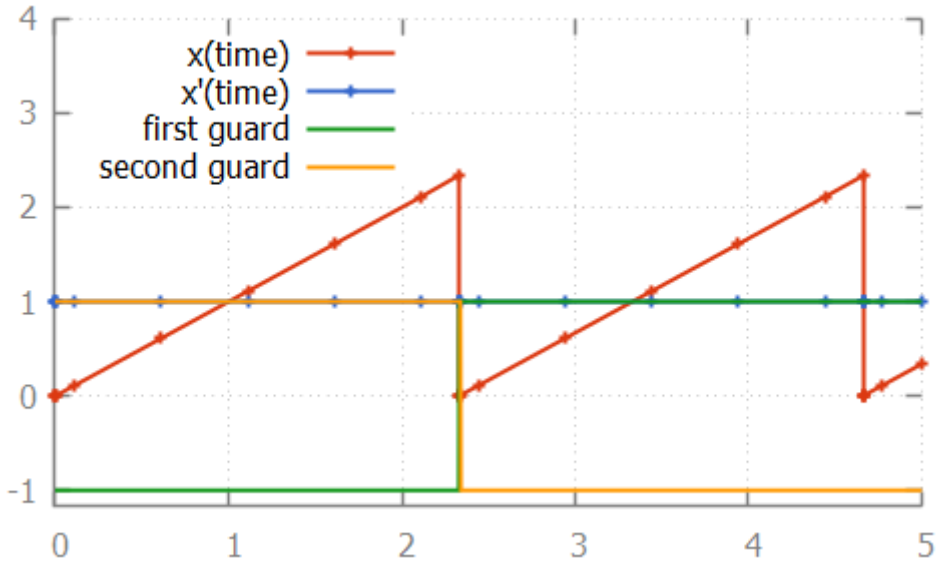
Solution 2: splitting the guard

We can reformulate the original CIF specification, by splitting the guard:

```
automaton p:
  cont x = 0.0;
  equation x' = 1.0;

  location:
    initial;
    edge when x >= 2.33, x <= 2.34 do x := 0.0;
end
```

This specification, when simulated using the CIF simulator, using default settings, results in:



The difference with the original specification is the guard of the edge. In the original specification, the `and` binary operator is used, making it a single guard. In the new specification, a comma (`,`) is used, which constructs two separate guards. Both guards can be individually checked by the root finding algorithm. As shown in the image, each of the guards changes value only once.

The comma between guards means that both guards have to hold, in order for the edge to be enabled. This is similar, but not identical, to the `and` binary operator. For the `and` binary operator, the left hand side must be evaluated before the right hand side, and the right hand side may only be evaluated if the left hand side evaluates to `true`. This is called short-circuit evaluation. For guards separated by commas, no such relation may be assumed. That is, even if the first guard evaluates to false, can the second guard still be evaluated by the root finding algorithm, for guard changes. It can even be the case that the second guard is evaluated before the first guard. These differences not only influence root finding, but are also of particular interest when the right hand side can result in a runtime evaluation failure. For instance, the following specification:

```
automaton p:
  cont x der 1.0;
  location:
    initial;
    edge when x > 0, 1 / x < 0.5 do x := 0.0;
end
```

results in a runtime error. The second guard (`1 / x < 0.5`) is evaluated even if the first guard (`x > 0`) does not hold. Initially, the first guard does not hold, and the second guard results in division by zero. Changing the comma between the guards to an `and` operator, makes it possible to simulate this specification.

Debug information

Debug information about what the ODE solver calculates, and for what time points, can be enabled using the *Debug output* option. This may be useful in figuring out why the guard was missed, and what settings need to be changed. See the Debug console output page for more information.

Fixed output step size

The integrator determines the time points for which values are calculated. By default, those points are included in the trajectories. This means that they are also the output of the trajectory data output component. The integrator determines the time points for which to calculate values, based on the complexity of the ODEs. The integrator can be influenced using the integrator options.

Sometimes however, it may be more convenient to have the integration output at more regular intervals. Using the *ODE solver fixed output step size* option (*ODE solver* category), a fixed output step size can be configured. This value must be an integer or real number larger than zero.

This option does not influence the actual time points for which the integrator calculates the values. Instead, it influences how these results are put in the trajectories, that are subsequently used by the simulator.

Setting a fixed output step size, ensures that the trajectories contain output for all multiples of the given step size. They will however also contain output for the start and end of each time transition, which may not be at multiples of the step size, due to guards becoming enabled or disabled at other times.

If real-time simulation is enabled, the default is changed to automatically use the model time delta that is used for real-time simulation, as the fixed output step size. This way, the trajectories contain the time points for which intermediate frames are generated. If real-time simulation is enabled, setting a custom fixed output step size is not supported.

Simulation end time and maximum delay

The maximum length of a time transition is determined by several factors:

- The current simulation time.
- The simulation end time.
- The maximum allowed delay amount.

The current simulation time is the value of variable `time`. Each time transition starts at the current time.

Using the *Simulation end time* option (*Simulator* category), the end time of the simulation can be configured. By setting a simulation end time, you ensure that when the value of variable `time` reaches that value, the simulation terminates. Settings a simulation end time also limits the maximum length of a time transition, since from the current time, a limited amount of time can pass before the end time is reached. By default the simulation end time is set to 'infinite', which means that simulation termination, and the maximum length of a time transition, are not limited by a simulation end time.

Using the *Maximum delay* option (*Simulator* category), the maximum delay amount for a single time transition can be set. If set, no time transitions longer than the specified value may be calculated. This limits the maximum length of time transitions to that value. Essentially, this cuts time transitions into parts with a maximum length specified by this option. By default, the maximum delay is set to 'infinite', which means that the maximum length of a time transition is not limited by

this option. Setting a finite maximum delay is mandatory when using the simulator's execution mode. In other situations, restricting the maximum delay amount for a time transition should generally be avoided. For debugging purposes however, it may be useful to disable passage of time altogether. This can be achieved by using a maximum delay of `0`.

If both a simulation end time and a maximum allowed delay amount are specified, the most restrictive of them determines the maximum length of the time transition. If neither is specified, the simulator will attempt to calculate a time transition of length `100.0`.

ODE solver integrator options

The integrator used by the ODE solver of the CIF simulator, can be configured using several options.

Integration algorithm

The *ODE solver integrator algorithm* option (*ODE solver: ODE integrator* category) can be used to configure the integrator algorithm to use. The following algorithms are available:

- Higham and Hall 5(4)
- Dormand-Prince 5(4)
- Dormand-Prince 8(5,3) (default)
- Gragg-Bulirsch-Stoer
- Adams-Bashforth (experimental)
- Adams-Moulton (experimental)

The algorithms differ in the complexity of the systems of ODEs that they can handle, their performance, the precision of their results, etc. All these algorithms use an adaptive step size.

Absolute and relative tolerance

The *ODE solver integrator absolute tolerance* option (*ODE solver: ODE integrator* category) and the *ODE solver integrator relative tolerance* option (*ODE solver: ODE integrator* category), can be used to configure how precise the results of the integrator should be. The lower the tolerance (or error), the higher the precision of the results, but also the more computing time it costs to get to that precision. The absolute precision is the difference between the prediction and the actual value, while the relative precision is that same difference, as a fraction of the actual value. In the latter case, precision reduces with increasing values.

Minimum and maximum step size

The interval between two consecutive time points calculated by the integrator can be configured using the *ODE solver integrator minimum step size* option (*ODE solver: ODE integrator* category) and

the *ODE solver integrator maximum step size* option (*ODE solver: ODE integrator* category).

The default minimum step size is `1e-15` and the default maximum step size is `0.5`. Note however that the last step can be smaller than the value specified for the minimum step size option.

Increasing the maximum default step size may lead to better performance, but can also lead to less precision, and root finding problems.

Number of steps

The *ODE solver integrator number of steps* option (*ODE solver: ODE integrator* category) can be used to configure the number of steps to use for the experimental Adams-Bashforth and Adams-Moulton integrators. The number of steps must be greater or equal to two.

ODE solver root finder options

The root finder used by the ODE solver of the CIF simulator, can be configured using several options.

Maximum check interval

The *ODE solver root finder maximum check interval* option (*ODE solver: ODE root finder* category) is explained on a separate page. See Problems with root finding.

Root finding algorithm

The *ODE solver root finding algorithm* option (*ODE solver: ODE root finder* category) can be used to configure the root finding algorithm to use. The following algorithms are available:

- Regula Falsi (False position) method
- Illinois method
- Pegasus method (default)

The *Illinois* and *Pegasus* methods are modified *Regula Falsi* methods. The algorithms differ only in how they choose the middle point of the interval. This influences their convergence speed. The *Regula Falsi* method in particular should be avoided in practice.

Absolute and relative tolerance

The *ODE solver root finder absolute tolerance* option (*ODE solver: ODE root finder* category) and the *ODE solver root finder relative tolerance* option (*ODE solver: ODE root finder* category), can be used to configure how precise the results of the root finder should be. The lower the tolerance (or error),

the higher the precision of the results, but also the more computing time it costs to get to that precision. The absolute precision is the difference between the prediction and the actual value, while the relative precision is that same difference, as a fraction of the actual value. In the latter case, precision reduces with increasing values.

Maximum iterations

The *ODE solver root finder maximum iterations* option (*ODE solver: ODE root finder* category) can be used to set the maximum number of iterations to use for root finding. This value must be at least one. If the root interval is larger than the tolerances allow, after the maximum number of iterations, root finding fails.

If it fails for your CIF specification, increase the value of this option. If that does not help, your specification is most likely too complex (by nature), or has a modeling error that causes the specification to become too complex for the root finding algorithm to handle. The problem however, may also be in one of the other root finder options, or in one of the integrator options.

Java compiler

The simulator internally generates Java code from the CIF model. This code is compiled using a Java compiler and executed. Code generation and compilation result in faster simulation. However, compilation itself also takes time. The time it takes to compile is different for different Java compilers.

Using the *Java Compiler* option (*Compiler* category), the Java compiler implementation to use, can be configured. We currently provide two different Java compilers:

- Java compiler from the Java Development Kit (`jdk`)

  The Java Development Kit (JDK) that is required to use our tooling includes a Java compiler. This compiler has the advantage that is has a bit less overhead of accessing the file system, while it has as disadvantage that it is a slower in actually compiling the Java code.

- Eclipse Compiler for Java (`eclipse`)

  The Eclipse Compiler for Java (ecj) is part of the Eclipse Java Development Tools (JDT), and is provided with our tooling. This compiler has the advantage that is a faster in actually compiling the Java code, while it has as disadvantage that it has a bit more overhead of accessing the file system.

The names given above, in parentheses, indicate the command line option values to use for selecting the different compilers.

In general, the JDK compiler is faster for small CIF models, while the Eclipse compiler is faster for larger CIF models. By default, the JDK compiler is used.

The *Java Compiler* option has no effect when simulating using a compiled code file (e.g. a `.cifcode` file) for repeated simulations, or when debugging generated code by enabling the *Load debug code* option (*Simulator* category).

Simulation miscellaneous topics

Execution mode

The CIF simulator's execution mode can be used to simulate a CIF specification in a way that adheres to the execution scheme defined by the CIF controller properties checker. This way the simulation will be more similar to the execution of generated code.

The simulator's execution mode can be used to reduce the validation effort, by simulating only traces that are similar to what will occur in executions of implementations of the specification. That is, traces that can't occur due to the execution scheme in executions, then also don't need to be simulated. However, while the simulator's execution mode tries to mimic the execution behavior as best it can, it may not be exactly the same. Enabling the execution mode thus runs the risk of missing behavior that can occur during executions.

The execution mode is disabled by default. It can be enabled using the *Execution mode* option (*Simulator* category).

Preconditions

To be able to enable the execution mode, several preconditions must be satisfied. Some preconditions are related to the specification being simulated:

- The specification being simulated must not have any 'tau' edges. This means both edges with explicitly a `tau` event on them, as well as edges without an event that are implicitly 'tau' edges, are not supported for execution mode. Such edges are explicitly forbidden by the execution scheme.
- The specification being simulated must not have an events that are neither controllable nor uncontrollable. Such events are explicitly forbidden by the execution scheme.
- The specification being simulated must not have any SVG input mappings that map to an event. Such SVG input mappings are not strictly compatible with the execution scheme. Instead, use SVG input mappings with updates.

Other preconditions are related to simulation options:

- A finite maximum delay for time transitions must be set. The execution scheme requires that the code to execute events is regularly executed. This means that the simulator must stop letting time pass regularly, allowing it to again consider event transitions. Therefore, a finite maximum duration of time transitions must be set.

  Typically, it is best to set this to a value that fits the execution frequency used for the execution of the implementation of the specification. If in the implementation the code is executed a certain number of times per second, so with a certain frequency, then the maximum delay should be set to match. For instance, for a frequency of 100 times per second, set the maximum delay to 1 / 100 = 0.01.

  If a maximum delay is chosen that is too large, then events may be executed much later than that they become enabled. For instance, if a transition for a controllable event leads to a state where an uncontrollable event becomes enabled, a transition for the uncontrollable event can

only occur after a time transition. Thus, it typically won't occur until 'maximum delay' time units have passed.

If real-time simulation is enabled, it is recommended to set the maximum delay to match the frame rate.

- The complete mode option must be disabled. The way the execution scheme is defined, calculating multiple different possible transitions from a single state is not applicable.

  Note that this also implies restrictions on which input mode can be used: only the automatic input mode and SVG input mode may be used. Furthermore, it implies that the automatic mode choice algorithm must be configured to always choose the first possible transition.

  By default, if these additional restrictions are satisfied, complete mode is automatically disabled, and therefore does not need to be disabled explicitly.

- The non-urgent events option must not be used to make any events non-urgent. That is, all events must be simulated as urgent. The way the execution scheme is defined, events are executed as part of their event loop as long as they are enabled. Time may only pass after the execution of both the uncontrollable and controllable event loops.

Working

The way the execution mode work, is as follows, starting from the first item for the initial state:

1. The simulator will try to perform transitions for SVG input mapping with updates, if any are queued due to having clicked a corresponding interactive SVG element. Once no more clicks need to be processed, it proceeds to the second item.
2. The simulator will try to perform transitions for uncontrollable events, in the order specified by the execution scheme. Once no more uncontrollable events are enabled, it proceeds to the third item.
3. The simulator will try to perform transitions for controllable events, in the order specified by the execution scheme. Once no more controllable events are enabled, it proceeds to the fourth item.
4. The simulator will try to perform a time transition. If no time transition is possible, it proceeds to the first item. If a time transition is possible, it proceeds to the first item after the time transition ends.

If from a state no transitions are possible for any of these items, the simulation deadlocks.

Complete mode

The CIF simulator calculates for each state the set of transitions that are possible from that state. It is up to the input component to choose one of those transitions, as the transition to take. Using the automatic input mode, it is possible to always choose the first transition that is possible.

In that situation, it is then a waste of resources to calculate the other possible transitions. Using the *Complete mode* option (*Simulator* category), it is possible to enable complete mode. If complete mode is enabled, the calculation of the possible transitions stops as soon as one possible transition

is found.

The following values are available for the complete mode option:

- *Auto: disable if possible, enable otherwise* (default)

  Automatically chooses whether to enable or disable complete mode. If it is allowed to disable complete mode, it will be disabled. Otherwise, complete mode will be enabled.

- *On: forced enabled*

  Enables complete mode. All possible transitions will be calculated.

- *Off: forced disabled*

  Disables complete mode. Only the first possible transitions will be calculated.

Currently, it is only allowed to disable complete mode, if the automatic input mode or SVG input mode is used, together with always choosing the first possible transition. Enabling complete mode is not supported when the execution mode is enabled.

External user-defined functions

The CIF simulator supports external user-defined functions. Currently, the following external languages are supported:

- Java

Termination

Termination requests are not processed during the execution of external user-defined functions. If the functions runs for a long time, the termination is postponed until the external function finishes its execution. If the execution never terminates, for instance because it gets stuck in a loop, often due to a bug, the termination request is never processed and the simulator will never terminate.

Side effects

All user-defined functions in CIF are pure mathematical functions. They must therefore be deterministic, and may not have side effects. For internal user-defined functions, this is enforced. For external user-defined functions, it is mostly impossible to check this constraint in an implementation, and the responsibility for checking this is therefore delegated to the end user. Practically, this means that for instance logging statements in functions, while essentially side effects, may be permitted, as long as the function returns the same value, if given the same arguments. This is essential for correct simulation results, as the results of function calls may for instance be cached by the simulator.

Value semantics

The CIF simulator uses value semantics for the parameters and return values of functions. That is, changes to the parameters of functions in external user-defined functions are *not* reflected back to the CIF specification. Furthermore, changes to the return values in the CIF simulator have no effect on subsequent calls to external user-defined functions.

External user-defined Java functions

External user-defined Java functions allow the use of static Java methods from CIF specifications. This page explains how to use such Java methods, how the simulator loads and invokes them, what is allowed, etc. It explicitly does not explain how to write or compile Java code.

External implementation reference

Consider the following examples of external user-defined Java function:

```
func real f1(real x): "java:java.lang.Math.expm1";

func int f2(bool x): "java:pkg.Cls.method|../some.jar;bin";
```

Two external user-defined functions are given as examples. Both have a string literal as external implementation reference. For Java functions, the string literal must start with `java:`. Then the absolute name of the Java method that is to be invoked is given. That is, the package name, class name, and method name are given, separated by dots (`.`).

Optionally, a Java class path may be given at the end, preceded by a bar (`|`) character. The class path entries must be separated by semicolons (`;`) regardless of the operating system. The class path entries are absolute or relative local file system paths to Jar files or directories with Java class files. Relative paths are resolved against the directory that contains the CIF specification in which the external user-defined function is declared. Both Linux/Mac file separators (`/`) and Windows file separators (`\`) may be used in class path entries. However, `\` must be escaped as `\\` in CIF string literals, and thus it is recommended to always use `/` as file separator. A discussion of the Java class path concept is beyond the scope of this documentation.

Supported types and values

Not all CIF data types are supported for the parameters and return values of external user-defined Java functions. The following table lists the allowed CIF types, and their corresponding Java types.

CIF type	Java type
`bool`	`java.lang.Boolean` / `boolean`
`int`	`java.lang.Integer` / `int`
`int[a..b]`	`java.lang.Integer` / `int`

CIF type	Java type
`real`	`java.lang.Double` / `double`
`string`	`java.lang.String`
`list t`	`java.util.List<t>`
`set t`	`java.util.List<t>`
`dict(k: v)`	`java.util.Map<k, v>`
`tuple(t1, t2, ..., tn)`	`java.util.List<java.lang.Object>`

Unless otherwise specified, the primitive Java types (`boolean`, `int`, and `double`) can be used interchangeably with their class variants (`java.lang.Boolean`, `java.lang.Integer`, and `java.lang.Double`). For lists and sets, the element type recursively affects the type parameters of the Java types. Similarly, for dictionaries, the key and value types affect the type parameters of the Java `Map` type. Since Java doesn't have tuples, and there is no way in Java to define a type with a variable number of type parameters, CIF tuples are passed to Java as a list of objects. It may be assumed that such lists have the same number of elements as there are fields in the original tuple, and that the elements have types that correspond to the types of the field of the tuple. The following table shows a few examples:

CIF type	Java type
`bool`	`boolean`
`int`	`int`
`list int`	`java.util.List<java.lang.Integer>`
`set tuple(int a, b)`	`java.util.Set<java.util.List<java.lang.Object>>`

For parameters, no `null` values are ever passed, not even as elements of lists, etc. Furthermore, for `real` typed values, `NaN`, positive infinity, and negative infinity are never passed along.

For return values, `null` values, as well as `NaN`, positive infinity, and negative infinity are considered runtime errors. Furthermore, integer values outside their CIF integer type range bounds are considered runtime errors as well.

Method resolution

The class as specified in the external implementation reference is loaded using a Java class loader. By default, the system class loader is used. If additional class path entries are specified in the external implementation reference, an extended class loader is used that searches in the additional class path entries before deferring to the system class loader. The class is loaded using the binary name of the class.

The method that is resolved must be a static method. The access modifier of the static method is irrelevant. That is, it doesn't matter whether the method is `private`, `protected`, `public`, etc.

The method is resolved in the Java Virtual Machine (JVM) that runs the simulator. The method is resolved by name, and parameter types. The parameter types only include the top level type. That

is, for CIF type `list int`, the Java type `java.util.List` is used as the type of the parameter. If more than one method with the same parameter types is declared in the class, and one of these methods has a return type that is more specific than any of the others, that method is used. Otherwise one of the methods is chosen arbitrarily.

For parameters that have primitive Java types as their type, the primitive types are used to resolve the method instead of their classes. In other words, when resolving the method, or the proper overload of the method, `int` and `java.lang.Integer` are considered incompatible.

The CIF simulator employs lazy loading. That is, if the external Java function is never called, it is also never loaded.

Method invocation

The method is invoked in the Java Virtual Machine (JVM) that runs the simulator, using reflection. The method is invoked synchronously, on the simulator thread.

Since the parameter types are not checked recursively when the method is resolved, methods may fail to execute properly if the type signatures don't match exactly. Similarly, for the return type only the top level type is checked. If a return value does not match the CIF type (wrong element type, wrong tuple length, etc), simulation will fail with a runtime error.

Any exceptions that occur during execution of the Java method are caught by the simulator. When the simulator catches such an exception, it will print the Java stack trace of the exception to the console, and simulation will be terminated.

Side effects

As all functions in CIF, external user-defined Java functions should not have any side effects. That is, if called with the same arguments, it should return the same value each time. More concretely, maintaining state information should be avoided, as should file I/O, and reading from stdin.

Writing to stdout and stderr should not affect the result of the method. It should however be avoided as well, as such writes are associated with the streams of the Java Virtual Machine (JVM), and not the simulator. The output will not end up on the Eclipse console of the simulator.

You should assume that each external function is loaded using a different class loader. As such, multiple Java methods from the same class, from the same external library, may be loaded using different class loaders as well. As such, shared static fields (class variables) of those classes can in general not be shared by different external functions.

Profiling

The CIF simulator has a *Profiling* option (*Simulator* category). If the option is enabled, information about the number of transitions taken per second, is printed to the console. Enabling profiling disables most other types of console output. By default, profiling is disabled.

The profiling output looks somewhat like this:

```
750431.42 transitions/second
640589.87 transitions/second
890939.01 transitions/second
887228.57 transitions/second
890143.25 transitions/second
893674.44 transitions/second
891812.69 transitions/second
```

Stochastic distributions and seeds

Stochastics can be used to model randomness in CIF models. The CIF simulator exhibits pseudo-randomness. This page explains how to influence that pseudo-randomness.

The randomness of the stochastic distributions is implemented in the CIF simulator using pseudorandom number generators (PRNGs). The PRNG used is the Mersenne Twister. The sequences of random numbers that are generated by the MT are not truly random. They are pseudorandom, as they start to repeat after some time. The Mersenne Twister however, has a very long *period* of 2^{19937} - 1 numbers.

Since the sequence of generated random numbers is always the same, the only way to vary it, is to use a different starting place. The starting place is determined by the initial seed, which can be configured through the *Initial seed for stochastic distributions* option (*Simulator* category). If specified, it must be a value in the range [1 .. 2^{30}]. If it is not specified, a random initial seed in that range is chosen by the simulator. If a random initial seed is chosen, and it is used, it is printed to the console.

The initial seed as provided by the option, is used for the first stochastic distribution that is created. Each subsequent distribution uses the next available seed. That is, if you specify an initial seed of 5, then the first distribution is created with an initial seed of 5, the next one uses 6, the third one uses 7, etc. The initial seeds are kept in the range [1 .. 2^{30}]. After seed 2^{30}, 1 is used as seed, then 2 is used as seed, etc.

Simulating the same specification multiple times, using the same initial seed, results in the same samples. Therefore, using a fixed seed can make your simulation deterministic, even though you use stochastic distributions to introduce randomness into your specification.

Simulation developer topics

Maximum time point tolerance



This page is primarily intended for the developers of the CIF simulator, and not its end users.

The CIF simulator internally often queries the trajectories computed by the ODE solver. Due to the limited precision of binary floating point number representations, requests of time points for

which no exact match is present in the trajectories, but for which a very close match is present, are allowed. The simulator compensates for these small deviations, to avoid accumulation of such deviations.

The simulator uses the value of the *Maximum time point tolerance* option (*Simulation* category) to determine whether two time points are close enough to each other to constitute a match. The tolerance is given in ulps (*units in the last place*, or *units of least precision*). One ulp is the difference between two subsequent floating point numbers, for the limited precision binary floating point number representation used by the simulator. That is, two floating point numbers are one ulp away from each other, if there is no other floating point number between them, that can be represented using the floating point number representation used by the simulator. Since an ulp increases as the floating point numbers increase, the tolerance is thus relative to the values of the time points.

The tolerance values chosen using the option must be in the range [1 .. 2^{30}]. The default tolerance is 2^{10} (= 1024) ulps. The default tolerance should suffice, but if it doesn't, you may be presented with an error like this:

```
ERROR: Simulation resulted in a runtime error, for state: time=0.0, x=1.0.
CAUSE: Requested time point "1.6000000000000005" is "2.0" ulps (1 ulp =
2.220446049250313e-16) away from the closest time point in the trajectories "1.6",
which is farther away than the maximum tolerance of "1" ulps. You can try increasing
the value of the maximum time point tolerance option. Please also contact the
development team.
```

If you get such an error, you can try to increase the tolerance using the *Maximum time point tolerance* option, in order to be able to continue your simulation. However, as this shouldn't actually happen, you should also contact the development team, who will then figure out why the default tolerance doesn't suffice for your model.

Debugging generated code



This page is primarily intended for the developers of the CIF simulator, and not its end users.

The CIF simulator generates highly optimized code for the specification, to speed up the simulation. By enabling the *Debug the generated code (by writing it to disk)* option (*Output* category), the generated code can be written to disk, for further debugging.

Output directory

If the simulation is performed from within Eclipse, and a project named `org.eclipse.escet.cif.simulator.debug` exists, is not closed, and has a `src` directory, the generated code will be written to that `src` directory. Otherwise, the generated code will be written to the directory containing the input specification file.

Regardless, the code is also always written to a `cifcode` sub-directory. If this sub-directory does not yet exist, it is created. If it already exists, all relevant existing files, such as `.java` files, are removed

from the sub-directory first, before the newly generated code is written.

Written files

This results in several files being written to the `cifcode` directory. One of these files is `DebugSimulator.java`, which contains the simulation options that were used and always shows the option dialog. The working directory for the simulation performed via `DebugSimulator.java` is the same working directory as the original simulation. This allows additional files, such as configuration files, to be found.

The `org.eclipse.escet.cif.simulator.debug` project contains a launch file (`debug-cif-simulator-generated-code.launch`) that can be used to reproduce the simulation implemented by the generated code. This launch configuration assumes that the code was generated in the `src` folder of the `org.eclipse.escet.cif.simulator.debug` project and was compiled to the `target/classes` folder of that project.

How to use

To use this feature, follow these steps:

- Start an Eclipse ESCET development environment.
- Make sure your workspace is automatically rebuilt, by enabling the **Build Automatically** option in the **Project** menu.
- Start a sub-Eclipse application, by clicking **Run › Run Configurations...** or **Run › Debug Configurations**, and selecting *escet-product* and clicking **[Run]**.
- In the sub-Eclipse that opens, make sure the CIF specification (the `.cif` or `.cifx` file) that you like to simulate is available in the workspace.
- In the sub-Eclipse, import the `org.eclipse.escet.cif.simulator.debug` source project.
- Simulate the CIF specification, and make sure you enable the *Debug the generated code (by writing it to disk)* option.
- Continue simulation, if desired.
- Once the simulation has terminated, switch back to the main Eclipse.
- Refresh the `org.eclipse.escet.cif.simulator.debug` project.
- Set breakpoints in the generated Java code, or in the simulator code, as desired.
- Right click `debug-cif-simulator-generated-code.launch` in the `org.eclipse.escet.cif.simulator.debug` project, and choose **Debug as › debug-cif-simulator-generated-code.launch**.
- Debug the simulation, with the generated code, as you would any Java program.

Starting the simulator with generated code

The launch configuration that is used, uses the *Load debug code* option (*Simulator* category). Using this option, the classpath (directory) where the already generated code is located, is specified. By

specifying it, the simulator knows from where to load it.

Simulations not started from this launch configuration, generally don't use this option. They load the input specification, validate it, generate fresh code, and compile the code in-memory, or use compiled code files.

Simulations using the generated code ignore all options related to repeated simulations. That is, the *Load debug code* option takes priority over the *Compile only* option (*Compiler* category), and simulating with compiled code files.

A warning is printed to the console if the generated code was generated with a different version of the simulator than the one used to simulate it. Ignoring the warning may lead to unexpected simulation results, or even simulator crashes.

Test mode



This page is primarily intended for the developers of the CIF simulator, and not its end users.

In order to be able to perform automated testing, the *Test mode* option (*Output* category) can be used. This option has the following effect:

- The visual interfaces for all output components (such as visualizers) are disabled.
- The user is no longer asked to confirm termination of the simulator, if the *Ask to confirm simulator termination* option (*Simulator* category) is configured to let the simulator decide whether to ask the user for confirmation or terminate automatically. By not asking for user input at the console, the automated test set can be fully automatically executed.
- It is allowed to perform real-time simulation without using any real-time output components, allowing testing of the real-time functionality in the automated test set.
- All real-time delays are skipped. By not performing unnecessary real-time delays, the automated test set executes as fast as possible.
- The SVG visualizer operates in synchronous mode, rather than asynchronous mode. This ensures that debug output becomes synchronous as well, and we thus get consistent output for our test cases. This is not only useful for automated testing, but also for debugging.

4.4.2. CIF to mCRL2 transformer

The CIF to mCRL2 transformer is used to convert a CIF specification to an mCRL2 model. mCRL2 is a tool for verifying properties of the model.

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF simulation, validation and verification tools › Convert CIF to mCRL2...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF simulation, validation and verification tools › Convert CIF to mCRL2...**.
- Use the `cif2mcrl2` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2mcrl2` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative file system path to the input CIF file.
- *Output file path*: The absolute or relative file system path to the output mCRL2 file. By default, the output file path is the same as the input file path, but with the `.cif` extension removed (if it exists), and the `.mcrl2` extension added. This option can be used to override the default.
- *Generate 'value' actions*: To query the value of a variable or current location of an automaton, a value action can be used. By default, such actions are generated for every discrete variable, input variable and automaton with at least two locations of the CIF specification. Using this option, you can define a reduced set of variables and automata that should get such an action. See the section on value actions for more details.
- *Generate 'marked' action*: Whether to generate a 'marked' action in the generated mCRL2 model. By default, a 'marked' action is generated. Using this option, generation of this action can be disabled. See the section on the marked action for more details.

Supported specifications

The CIF to mCRL2 transformer supports a subset of CIF specifications.

- Regarding variables:
  - Discrete variables are supported. The following restrictions apply:
    - They must have a `bool`, `enum`, or `int` type. The latter may have a range.
    - The initial values of a variable, if explicitly specified, must be a statically evaluable.
    - The transformation ensures that integer variables remain within their range, thus preventing runtime errors.
  - Continuous variables are not supported.
  - Algebraic variables are supported, if their types and values are supported.
- Regarding automata:
  - There must be at least one automaton.

    - The supervisory kind of automata is ignored.
    - Multi-assignments are not supported. Multiple assignments on an edge is supported.
    - Urgent locations and edges are not supported.
- Regarding expressions:
    - All expressions and sub-expressions must be of type `bool`, `enum`, or `int`. The latter may have a range.
    - Regarding boolean-typed expressions (resulting in a boolean as value):
        - Boolean literals (`false` and `true`) are supported.
        - Boolean-typed constants are supported.
        - Boolean-typed discrete variables are supported.
        - Boolean-typed input variables are supported.
        - Boolean-typed algebraic variables are supported.
        - Boolean-typed received values.
        - Binary operators `and`, `or`, `=>` and `<=>` on boolean-typed arguments are supported.
        - Binary operators `=` and `!=` on boolean, enum and integer-typed arguments are supported.
        - Binary operators `<`, `<=`, `>` and `>=` on integer-typed arguments are supported.
        - Unary operator `not` on boolean-typed arguments is supported.
        - Location references are supported.
        - `if` expressions with a boolean-typed result are supported.
        - `switch` expressions with a boolean-typed result, and supported `switch` and key values, are supported.
        - Cast expressions on boolean-typed arguments that don't change the type are supported.
    - Regarding enumeration-typed expressions (resulting in an enumeration literal as value):
        - Enumeration literals are supported.
        - Enumeration-typed constants are supported.
        - Enumeration-typed discrete variables are supported.
        - Enumeration-typed input variables are supported.
        - Enumeration-typed algebraic variables are supported.
        - Enumeration-typed received values.
        - `if` expressions with an enumeration-typed result are supported.
        - `switch` expressions with an enumeration-typed result, and supported `switch` and key values, are supported.
        - Cast expressions on enumeration-typed arguments that don't change the type are supported.
    - Regarding integer-typed expressions (resulting in an integer as value):
        - Integer literals are supported.

        - Integer-typed constants are supported.
        - Integer-typed discrete variables are supported.
        - Integer-typed input variables are supported.
        - Integer-typed algebraic variables are supported.
        - Integer-typed received values.
        - Binary operators `+`, `*` and `-` on integer-typed arguments are supported.
        - Binary operators `div` and `mod` on positive divisors are supported.
        - Unary operators `-` and `+` on integer-typed arguments are supported.
        - `if` expressions with an integer-typed result are supported.
        - `switch` expressions with an integer-typed result, and supported `switch` and key values, are supported.
        - Cast expressions on integer-typed arguments that don't change the type are supported.
- Regarding other concepts:
    - Requirement state invariants are not supported.
    - State invariants that depend on input variables are not supported.
    - I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
    - Annotations are ignored.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove I/O declarations
- Remove annotations
- Eliminate component definition/instantiation
- Simplify values

The following transformations are additionally applied as preprocessing (in the given order), after checking preconditions, to simplify the transformation:

- Eliminate type declarations
- Eliminate constants
- Eliminate algebraic variables
- Eliminate state/event exclusion invariants
- Linearize (product with merge)
- Eliminate state invariants

- Convert `switch` expressions to `if` expressions
- Eliminate `if` updates

Output of the transformation

Names

The mCRL2 names of CIF objects, such as automata, variables and events, are based on the absolute names of the CIF objects. Each `.` is replaced by an apostrophe (`'`), to form a valid mCRL2 name. A single apostrophe is added at the end of each name, to ensure that the names don't conflict with mCRL2 keywords. For an automaton `a` with a location `x`, its absolute name is `a.x`. Its mCRL2 name is thus `a'x'`.

Enumerations

The generated mCRL2 model starts with an mCRL2 sort for each CIF enumeration. For compatible enumerations, a single representative is used. Due to linearization, locations of automata with at least two locations are represented as location pointer variables, and their values come from enumerations as well.

Events

For each CIF event, an mCRL2 action is generated.

Input variables

For each CIF input variable, an mCRL2 action is generated. Such an action is used to allow an input variable to change value. The name of the action is the mCRL2 name of the input variable, with `'input` added to it as postfix.

Variable 'value' actions

For each CIF discrete and input variable, a 'value' action may be generated. Due to linearization, locations of automata with at least two locations are represented as location pointer variables, and thus an action may be generated for them as well. The name of the action is the mCRL2 name of the variable, with `'varvalue` added to it as postfix. These mCRL2 actions have as type the type of the corresponding CIF variable.

By default, a 'value' action is generated for each discrete variable, each input variable, and each automaton with at least two locations. In some cases, this may not be wanted. Variables that never have to be queried for their value do not need such an action. Having such an action available anyway increases the amount of work that has to be done by the mCRL2 tool. To restrict the generation of 'value' actions, the *Generate 'value' actions* option can be used (`--value-actions=⋯` or `-v ⋯`).

The option takes a comma separated list of variable/automata name patterns, that are considered in the order they are given. Each pattern can add or remove variables and automata. If the pattern starts with a `+` character, variables are added and thus get a 'value' action. If the pattern starts with

a - character, variables are removed and thus don't get a 'value' action. If the pattern start neither with a + nor with a - character, variables are added.

The main part of each pattern is the name of the variable or automaton to add or remove. The absolute names of variables and automata should be used. Only automata with at least two locations can be used. In addition, you can use the * character as a shorthand for 'zero or more arbitrary characters', allowing you to write abbreviations, and match several variables and/or automata at the same time.

The list of patterns is interpreted relative to selecting no variables. That is, if an empty list of patterns is specified, no variables and automata are added. By default, the +* pattern is used, which adds all variables.

For example, assume existence of the following automata and variables in a CIF specification:

```
A.p12 A.q2 A.z B B.x B.y1 B.y2 B.z C.q
```

Below a number of example option values are given, together with what automata and variables they select from the above list.

- The option value B.x,+A.z selects variables B.x and A.z.
- The option value +* selects all automata and variables.
- The 'empty' option value selects no automata and variables. On the command line or in ToolDef scripts, when using the -v short option, use '' for the 'empty' option value.
- The option value +*1 selects all automata and variables that end with a 1, which in this case is variable B.y1 only. Variable A.p12 does have a 1 in it, but not at the end.
- The option value +*1* selects all automata and variables with a 1 at any position. This matches both A.p12 and B.y1.
- The option value +*,-B,-B.* selects all automata and variables, except automaton B and the variables defined in it. First, +* adds all automata and variables. Then, -B removes automaton B. Finally, -B* removes all variables that start with B., so all variables in automaton B.

'marked' action

By default, a marked action is generated. The action is enabled when the CIF specification's marker predicate holds, and is disabled otherwise. The action allows to verify properties in mCRL2 that relate to marked states of the CIF specification. To disable the generated of this action, use the *Generate 'marked' action* option can be used (--marked-action=off or -a0).

Behavior

A single mCRL2 process named P is generated that represents the behavior of the CIF specification. The process has a parameter for each variable of the CIF specification. Due to linearization, this includes the location pointer variables for automata with at least two locations.

The process consists of alternative process expressions. For each linearized CIF edge, there is one alternative. Each of these alternatives consists of a guard, an action and a recursive invocation of

process `P` with the updates to the variables. The mCRL2 guard expressions are constructed from the guards of the linearized CIF edges. Additional guards are added to ensure that integer variables remain within their ranges, thus preventing runtime errors.

For each CIF input variable, and additional alternative process expression is added. It allows the input variable to change to a different value, so to any value supported by its type, except for the value it already has.

For each CIF variable and automaton with a variable 'value' action, an additional alternative process expression is added. It allows the variable 'value' action to occur for the current value of the variable.

Initialization

At the end of the generated mCRL2 program, an initial mCRL2 process is given. It matches the initial state of the CIF specification.

If the CIF specification has multiple initial states and/or initialization predicates in components (so outside locations), an `initialize` action is generated in the mCRL2 program. The `initialize` action is performed to determine the initial state, after which process `P` is started from that initial state. No `initialize` action is generated if the CIF specification has only one initial state and does not have any initialization predicates, including no initialization predicates generated by linearization. If the CIF specification has multiple initial states, the `initialize` action non-deterministically chooses one of them, allowing verification using mCRL2 to consider the behavior of the CIF specification from any initial state. If the CIF specification has no initial state, then the mCRL2 program always has an `initialize` action, but it is not possible to take that action, and thus the process deadlocks immediately.

Example

Consider the following CIF specification:

```
event a, b, c;

automaton P:
  disc int[0..10] x = 0;
  disc int[-5..5] zero = 0;

  location p1:
    initial;
    marked;
    edge a do x := x + 1 goto p2;

  location p2:
    edge c when Q.y != 0 goto p1;
    edge a do x := 1 goto p1;
end

automaton Q:
  disc int[0..1] y = 1;
```

```
  location q1:
    initial;
    edge a when y = 1 goto q2;

  location q2:
    marked;
    edge b when y = 1 and P.x < 8 and P.zero = 0 goto q1;
    edge b when y = 0 => P.x >= 8 goto q1;
end
```

By default, the following mCRL2 model is generated for it:

```
% Generated by the CIF to mCRL2 transformer of the Eclipse ESCET toolkit.

% Sorts for CIF enumerations.
sort P'LPE' = struct P'p1' | P'p2';
sort Q'LPE' = struct Q'q1' | Q'q2';

% Actions for CIF events.
act a';
act b';
act c';

% Actions for CIF variables/automata having certain values/locations.
act P''varvalue: P'LPE';
act P'x''varvalue: Int;
act P'zero''varvalue: Int;
act Q''varvalue: Q'LPE';
act Q'y''varvalue: Int;

% Action for CIF marker predicate.
act marked;

% Process for behavior of the CIF specification.
proc P(
    P'x': Int,
    P'zero': Int,
    P': P'LPE',
    Q'y': Int,
    Q': Q'LPE'
) =
    % CIF linearized edges.
    (((((P' == P'p1') || (P' == P'p2')) && (Q' == Q'q1')) && ((Q'y' == 1) && ((0 <=
if((P' == P'p1'), (P'x' + 1), if((P' == P'p2'), 1, P'x'))) && (if((P' == P'p1'), (P'x'
+ 1), if((P' == P'p2'), 1, P'x')) <= 10))))) -> a' . P(P'x' = if((P' == P'p1'), (P'x'
+ 1), if((P' == P'p2'), 1, P'x')), P' = if((P' == P'p1'), P'p2', if((P' == P'p2'),
P'p1', P')), Q' = Q'q2')
    +
    ((((Q' == Q'q2') && (Q'y' == 1)) && ((P'x' < 8) && (P'zero' == 0)))) -> b' . P(Q'
```

```
= Q'q1')
    +
    (((Q' == Q'q2') && ((Q'y' == 0) => (P'x' >= 8)))) -> b' . P(Q' = Q'q1')
    +
    (((P' == P'p2') && (Q'y' != 0))) -> c' . P(P' = P'p1')
    +
    % CIF variable value actions.
    P''varvalue(P') . P()
    +
    P'x''varvalue(P'x') . P()
    +
    P'zero''varvalue(P'zero') . P()
    +
    Q''varvalue(Q') . P()
    +
    Q'y''varvalue(Q'y') . P()
    +
    % CIF 'marked' action.
    (((P' == P'p1') && (Q' == Q'q2'))) -> marked . P()
;

% Initialization.
init P(0, 0, P'p1', 1, Q'q1');
```

4.4.3. CIF to UPPAAL transformer

The CIF to UPPAAL transformer can be used to transform CIF specifications to UPPAAL systems (`*.xml` files). UPPAAL is a tool modeling, validation, and verification of networks of timed automata with variables. The transformer only transforms untimed CIF specifications.

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF simulation, validation and verification tools › Convert CIF to UPPAAL...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF simulation, validation and verification tools › Convert CIF to UPPAAL...**.
- Use the `cif2uppaal` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2uppaal` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output UPPAAL file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.xml` file extension is added.

Supported specifications

The CIF to UPPAAL transformer supports a subset of CIF specifications. The following restrictions apply:

- Specifications without automata are not supported.
- Channels (events with data types) are not supported.
- Initialization predicates in components are not supported.
- Automata that do not have exactly one initial location are not supported.
- Locations with initialization predicates that are too complex to evaluate statically, are not supported. That is, those predicates must essentially be constant. For instance, `true` and `true or false` are supported, as is `c` if `c` is a constant. However, `v => c` with `v` a discrete variable that can initially have several different values, is not supported.
- Discrete variables with multiple potential initial values are not supported.
- Discrete variables with initial values that are too complex to evaluate statically, are not supported. That is, their initial values must essentially be constant. For instance, `1 + 1` and `2 * 5` are supported, as is `c + 1` if `c` is a constant. However, `v * 2` with `v` a discrete variable that can initially have several different values, is not supported.
- Continuous variables are not supported.
- Input variables are not supported.
- User-defined functions are not supported.
- Urgent edges are not supported.
- Multi-assignments on edges (such as `do (x, y) := (1, 2)`) are not supported. However, it is allowed to use multiple assignments on an edge (such as `do x := 1, y := 2`).
- Partial variable assignments (such as `do x[0] := 5`) are not supported.
- Conditional updates (`if` updates) on edges (such as `do if b: x := 5 end`) are not supported.
- Only the following data types are supported: boolean, integer (both with a range and without a range), and enumeration types.
- Only the following expressions are supported: boolean literal values (`true` and `false`), integer literal values, enumeration literal values, binary expressions (partially, see below), unary expressions (partially, see below), casts that don't change the type, `if` expressions, and references to constants, discrete variables, algebraic variables, and locations.

- Only the following binary operators are supported: logical equivalence (`<=>`), logical implication (`=>`), logical conjunction (`and` on boolean operands), logical disjunction (`or` on boolean operands), addition (`+`) on integer operands, subtraction (`-`) on integer operands, multiplication (`*`) on integer operands, integer division (`div`), integer modulus (`mod`), equality (`=`) on integer and enumeration operands, inequality (`!=`) on integer and enumeration operands, less than (`<`) on integer operands, less than or equal to (`<=`) on integer operands, greater than (`>`) on integer operands, and greater than or equal to (`>=`) on integer operands.
- Only the following unary operators are supported: logical inverse (`not`), negation (`-`) on an integer operand, and plus (`+`) on an integer operand.
- The controllability of events is ignored.
- The supervisory kinds of automata are ignored.
- The supervisory kinds of invariants are ignored.
- Marker predicates are ignored.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
- Annotations are ignored.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove I/O declarations
- Remove annotations
- Eliminate component definition/instantiation
- Eliminate `tau` event
- Eliminate state/event exclusion invariants
- Eliminate algebraic variables
- Convert enumerations to integers
- Eliminate monitors

Transformation result

CIF features synchronizing events, while UPPAAL only supports channels. A `SendAut` automaton/template is added to the UPPAAL system to ensure proper event synchronization. The UPPAAL template has a single location, and self loops for every event in the CIF specification. The guards of the self loops express the conditions (from edge guards and state/event exclusion invariants) under which the events are globally enabled (guard wise) in the CIF specification. Every CIF event is a broadcast channel in UPPAAL. If the self loop is enabled (guard wise), the `SendAut` template broadcasts over the channel. All the other templates (for the CIF automata) receive the event. They can actually receive, as the guard of the self loop ensures that. Together the `SendAut` self

loops and the receive edges form the synchronization.

As the `SendAut` template needs to refer to locations of the other templates, location pointer variables are added for all the other templates. For every CIF automaton `some.aut`, a location pointer variable `LP_some_aut` is added. The location pointers are integer variables with range `[0,n-1]`, for an automaton with `n` locations. Updates are added to the edges to ensure the location pointer variables have the proper values. The location pointers are similar to those created by the CIF to CIF transformation that eliminates the use of locations in expressions.

In CIF, assignments interpret the right hand side of the assignment (the new value of the variable) in the source state of the transition. In UPPAAL, assignments have order, and the right hand sides are interpreted over the current state, after any preceding assignments. Furthermore, the order in which the assignments of the edges of different participating templates are executed is not defined. To ensure the proper CIF semantics, 'old' versions of all variables (including the location pointer variables) are added. For a variable `x`, `OLD_x` is added. The `SendAut` automaton assigns the current values of all variables to their 'old' counterparts. The assignments on the edges of the other automata then use the 'old' variables to compute the new values of the variables. As the values of the 'old' variables are only used during the transitions, the 'old' variables are *meta* variables in the UPPAAL system.

The state invariants from CIF components are added to the location of the `SendAut` template.

For CIF variables with an `int` type, the UPPAAL type is `int[-2147483648,2147483647]`.

The names of templates, variables, etc in UPPAAL are based on the absolute names of their CIF counterparts. For a variable `a.b.c` in CIF, the UPPAAL name is `a_b_c`. If there are conflicts between the UPPAAL names, or if one of the UPPAAL names conflicts with a UPPAAL keyword, renaming is performed, and a warning is printed to the console.

No geometry is generated. When the generated UPPAAL file is opened in UPPAAL, UPPAAL will perform some layouting.

4.4.4. Controller properties checker

The controller properties checker can be used to determine whether a supervisor to be implemented as a controller satisfies certain desired properties. These properties are not yet guaranteed through synthesis, and must therefore be checked on a synthesized supervisor, before code is generated from it.

The controller properties checker assumes a certain execution scheme for code generated from CIF specifications by code generators. This scheme prescribes, among others, the use of iterations of the event loops and the order in which transitions are executed. All code generators must adhere to this scheme, for the results of the controller properties checker to carry over to implemented controllers.

The following properties can be checked:

Bounded response

The bounded response property holds if for each execution of the code, the number of iterations of the event loops to execute for uncontrollable and controllable events are both bounded. This

property ensures that the code that executes the uncontrollable events, as well as the code that executes the controllable events, always terminate. That is, the system has no loops with only uncontrollable events, or with only controllable events. In other words, there is no livelock for uncontrollable events, nor for controllable events.

This check does not produce false negatives.

Read more about this check on the bounded response check page.

Confluence

The confluence property holds if for each execution of the controllable events, it does not matter in which order the controllable events are executed, as the same state is reached regardless. This property ensures that the controller can make any choice between controllable events without losing the guarantees of synthesis. By extension, this means that code generators may generate code for the controller that executes the controllable events in any order.

This check does not check for confluence of the uncontrollable events. Furthermore, the check may produce false negatives.

Read more about this check on the confluence check page.

Finite response

The finite response property holds if for each execution of the code, the number of transitions to execute for controllable events is finite. This property ensures that the code that executes the controllable events always terminates. That is, the system has no loops with only controllable events. In other words, there is no livelock for controllable events.

This check does not check for finite response of the uncontrollable events. Furthermore, the check may produce false negatives. Hence, it is recommended to use the bounded response check instead.

Read more about this check on the finite response check page.

Non-blocking under control

The non-blocking under control property holds if a marked state can be reached, even though the transitions for uncontrollable events and controllable events are executed in a specific way in generated code. This property ensures that the code is still non-blocking, and this guarantee of synthesis is thus preserved in generated code.

This check does not produce false negatives.

Read more about this check on the non-blocking under control check page.

After performing the checks, the controller properties checker adapts the input model, to encode the results of the checks that were performed in the controller properties annotation of the model. It then writes the modified model to a file. If only some of the checks are performed, the earlier results of checks that are skipped are preserved. This allows the checks to be performed one by one.

After checking whether the properties hold on a model, that model should not be adapted anymore, as this may invalidate the results of the checks. If changes must be made, such as adding timers

before generating code, then such changes should be made with great care, as not to invalidate the results of the checks.

Starting the checker

The checker can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF simulation, validation and verification tools › Apply controller checks...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF simulation, validation and verification tools › Apply controller checks...**.
- Use the `cifcontrollercheck` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifcontrollercheck` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative file system path to the input CIF specification.
- *Enable bounded response checking*: If set (the default), bounded response is checked in the CIF specification. If unset, bounded response checking is not performed.
- *Enable confluence checking*: If set (the default), confluence is checked in the CIF specification. If unset, confluence checking is not performed.
- *Enable finite response checking*: If set (the default), finite response is checked in the CIF specification. If unset, finite response checking is not performed.
- *Enable non-blocking under control checking*: If set (the default), non-blocking under control is checked in the CIF specification. If unset, non-blocking under control checking is not performed.
- *Maximum number of printed cycle states*: The maximum number of *printed cycle states* to print to the console, in case bounded response doesn't hold due to cycles. The integer number must be non-negative and at most 2,147,483,639. By default, the limit is 10 printed cycle states. For further details, see the bounded response check documentation.
- *Print control loops*: If set (the default), during finite response checking the events that may still occur in a controllable-event loop are printed to the console.
- *Output file*: The absolute or relative file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.checked.cif` file extension is added.

At least one of the checks must be enabled.

Supported specifications

The CIF controller properties checker supports a subset of CIF specifications. The following restrictions apply for the tool in general, regardless of which checks are performed:

- Specifications without automata are not supported.
- Automata with non-determinism for controllable events are not supported. That is, automata that have locations with multiple outgoing edges for the same controllable event, with overlapping guards (e.g. `x > 1` and `x < 4`), are not supported. Note that this check may lead to false positives, the check is an over-approximation and guard overlap may be detected for unreachable states.
- Only plant and supervisor automata are supported. Kindless and requirement automata are not supported.
- Only plant and supervisor invariants are supported. Kindless and requirement invariants are not supported.
- State invariants are not supported, unless they are trivially `true`.
- Events not declared as controllable or uncontrollable are not supported. This includes the `tau` event, both explicitly used on edges, as well as implicitly for edges without explicitly mentioned events.
- Continuous variables are not supported.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
- Annotations are ignored.

The following restrictions apply only to the bounded response, confluence and non-blocking under control checks:

- Discrete and input variables are supported, but only if they have a supported type:
  - Boolean type.
  - Ranged integer type (e.g. `int[0..5]`). Ranges that allow more than 2,147,483,647 (2^31 - 1) possible values are not supported.
  - Enumeration type.
  - Tuple type, with supported types for the tuple's fields.
- Discrete variables must have supported initial values. If explicit initial values are given, they must be supported predicates (for boolean variables) or supported expressions as described below (for all other variables).
- Constants, algebraic variables and algebraic parameters of components are supported as long as their values are supported in the context where they are used (as described below).
- The following restrictions apply to user-defined functions:
  - External user-defined functions are not supported.
  - Internal user-defined functions with recursive calls are not supported.
  - Internal user-defined functions with `while` statements are not supported.

    - Internal user-defined functions with assignments that may result in runtime out-of-range errors are not supported.
    - Internal user-defined functions may only be used in function calls. All other uses are not supported, including using them as data, such as storing them in variables, returning them from functions, and so on.
    - Internal user-defined functions may only use supported data types (same as discrete and input variables) as their return and parameter types, and as the types of their local variables.
- Only limited forms of predicates (for markers, invariants, initialization, guards, initial values of boolean variables, right hand sides of assignments, etc) are supported. The supported predicates are:
    - Boolean literals (`true`, `false`).
    - Discrete and input boolean variables (`x`, for `x` a discrete or input variable with a boolean type).
    - Algebraic boolean variables with supported predicates as their values (`x`, for `x` an algebraic variable with a boolean type).
    - Boolean constants (`x`, for `x` a constant with a boolean type).
    - Locations (`aut.loc`, for `aut` and automaton and `loc` a location of that automaton).
    - The unary inverse operator (`not`) on a supported predicate.
    - The binary logical conjunction (`and`) on two supported predicates.
    - The binary logical disjunction (`or`) on two supported predicates.
    - The binary logical implication (`=>`) on two supported predicates.
    - The binary logical bi-conditional (`<=>`) on two supported predicates.
    - The binary equality comparison (`=`) on two supported predicates, on two supported integer or enumeration expressions, or on two supported tuple expressions.
    - The binary inequality comparison (`!=`) on two supported predicates, on two supported integer or enumeration expressions, or on two supported tuple expressions.
    - The binary less than comparison (`<`) on two supported integer expressions.
    - The binary less than or equal comparison (`<=`) on two supported integer expressions.
    - The binary greater than comparison (`>`) on two supported integer expressions.
    - The binary greater than or equal comparison (`>=`) on two supported integer expressions.
    - Conditional expressions (`if` expressions) with supported guard and resulting value predicates.
    - `Switch` expressions with supported control value and resulting value predicates.
    - Function calls to internal user-defined functions with supported arguments that return a boolean value.
    - Projections on tuples that result in a boolean value.
    - Received values (`?`) with a boolean type.

- Any other valid CIF predicate (computation) that results in a boolean value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

  Here are some examples of computations that can be statically evaluated:

  - `{1} in {1, 2}` (result is `true`)
  - `[false, true][0]` (result is `false`)
  - `c - 1`, for a constant `c` with value `4` (result is `3`)

  Here are some examples of computations that can *not* be statically evaluated:

  - `v = true` for `v` a discrete variable. The computation results in different values for different values of `v`.
  - `{v} in {1, 2}` for `v` a discrete variable. The computation results in different values for different values of `v`.

- Only limited forms of integer and enumeration expressions (for binary comparisons, initial values of variables, right hand sides of assignments, etc) are supported. The supported expressions are:
  - An integer literal/value.
  - An enumeration literal/value.
  - Discrete and input integer/enumeration variables (`x`, for `x` a discrete or input variable with an integer or enumeration type).
  - Algebraic integer/enumeration variables with supported expressions as their values (`x`, for `x` an algebraic variable with an integer or enumeration type).
  - Integer/enumeration constants (`x`, for `x` a constant with an integer or enumeration type).
  - `+i` and `-i` for `i` a supported integer expression.
  - `i + j`, `i - j`, and `i * j` for `i` and `j` supported integer expressions.
  - `i div j` and `i mod j` for `i` a supported integer expressions, and `j` a positive integer value, or a computation that results in a positive integer value, as long as the computation is not too complex to be performed statically. That is, `j` must essentially be constant.
  - `abs(i)` and `sign(i)` for `i` a supported integer expression.
  - `max(i, j)` and `min(i, j)` for `i` and `j` supported integer expressions.
  - Conditional expressions (`if` expressions) with supported guard predicates and supported resulting values.
  - `Switch` expressions with supported control values and supported resulting values.
  - Function calls to internal user-defined functions with supported arguments that return an integer or enumeration value.
  - Projections on tuples that result in an integer or enumeration value.
  - Received values (`?`) with an integer or enumeration type.
  - Any other valid CIF expression (computation) that results in an integer value or an

enumeration value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

Here are some examples of computations that can be statically evaluated:

- `2 * 5` (result is `10`)
- `floor(3.14)` (result is `3`)
- `c + 1`, for a constant `c` with value `2` (result is `3`)

Here are some examples of computations that can *not* be statically evaluated:

- `v - 1`, for `v` a discrete variable. The computation results in different values for different values of `v`.
- `ceil(sqrt(-1.0))`, since `sqrt` of a negative number results in a runtime error upon evaluation.

- Only limited forms of tuple expressions (for binary comparisons, initial values of variables, right hand sides of assignments, etc) are supported. The supported expressions are:
  - A tuple literal/value.
  - Discrete and input tuple-typed variables (`x`, for `x` a discrete or input variable with a tuple type).
  - Algebraic tuple-typed variables with supported expressions as their values (`x`, for `x` an algebraic variable with a tuple type).
  - Tuple-typed constants (`x`, for `x` a constant with a tuple type).
  - Conditional expressions (`if` expressions) with supported guard predicates and supported resulting tuple-typed values.
  - `Switch` expressions with supported control values and supported resulting tuple-typed values.
  - Function calls to internal user-defined functions with supported arguments that return a tuple value.
  - Projections on tuples that result in a tuple value.
  - A received value (`?`) with a tuple type.
  - Any other valid CIF expression (computation) that results in a tuple value, as long as the computation is not too complex to be performed statically. That is, the computation must essentially represent a fixed/constant value.

The following restrictions apply only to the finite response check:

- Channels (events with data types) are not supported.
- Functions (both standard library functions and user-defined ones) are not supported.
- Only discrete, input and algebraic variables with a boolean, ranged integer (e.g. `int[0..5]`), or enumeration type are supported.
- Only the following expressions are supported: boolean literal values (`true` and `false`), integer

literal values, enumeration literal values, binary expressions (partially, see below), unary expressions (partially, see below), casts that don't change the type, `if` expressions, `switch` expressions, and references to constants, discrete variables, input variables, algebraic variables, and locations.

- Only the following binary operators are supported: logical equivalence (`<=>`), logical implication (`=>`), logical conjunction (`and` on boolean operands), logical disjunction (`or` on boolean operands), addition (`+`) on integer operands, subtraction (`-`) on integer operands, multiplication (`*`) on integer operands, integer division (`div`), integer modulus (`mod`), equality (`=`) on boolean, integer or enumeration operands, inequality (`!=`) on boolean, integer or enumeration operands, less than (`<`) on integer operands, less than or equal to (`<=`) on integer operands, greater than (`>`) on integer operands, and greater than or equal to (`>=`) on integer operands.
- Only the following unary operators are supported: logical inverse (`not`), negation (`-`) on an integer operand, and plus (`+`) on an integer operand.
- Multi-assignments on edges (such as `do (x, y) := (1, 2)`) are not supported. However, it is allowed to use multiple assignments on an edge (such as `do x := 1, y := 2`).
- Partial variable assignments on edges (such as `do x[0] := 5`) are not supported.

Preprocessing

To increase the subset of CIF specifications that can be checked, various CIF to CIF transformations are applied as preprocessing.

The following transformations are applied for the tool in general, regardless of which checks are performed:

- Eliminate component definition/instantiation

Any changes they cause are included in the output model as well.

The following transformations are applied for the tool in general, regardless of which checks are performed:

- Remove I/O declarations

Any changes they cause are not included in the output model.

The following transformations are applied, in case bounded response or non-blocking under control is checked:

- Relabel supervisors as plants

Any changes they cause are not included in the output model.

The following transformations are applied (in the given order), in case confluence or finite response is checked:

- Eliminate state/event exclusion invariants

- Eliminate monitors
- Eliminate automaton `self` references
- Eliminate type declarations
- Eliminate the use of locations in expressions
- Convert enumerations to integers
- Eliminate algebraic variables
- Eliminate constants
- Simplify values
- Eliminate `if` updates

Any changes they cause are not included in the output model.

Execution scheme

The controller properties checker assumes a certain execution scheme for code generated from CIF specifications by code generators. This scheme prescribes, among others, the order in which transitions are executed. All code generators must adhere to this scheme, for the results of the controller properties checker to carry over to implemented controllers.

Assumptions

The scheme assumes that the CIF specification has only controllable and uncontrollable events. It thus assumes that the specification has no kindless event. This includes the absence of any `tau` events, either explicitly on edges or implicitly by the absence of event on edges.

Furthermore, the description of the scheme below assumes that the specification has no component definitions/instantiations. If the specification has them, they must be eliminated using the Eliminate component definition/instantiation CIF to CIF transformation. By always using this transformation, we ensure that all tools eliminate them in the same way.

Code parts

From a CIF specification, code can be generated using code generators. It is assumed that the code is regularly executed, for instance at a fixed or variable time interval. The code execution must be completed before the same code is executed again. For instance, if PLC code is generated, the main program would be executed once per PLC cycle.

The code has three parts, that are executed in order:

1. The values of the input variables are read from the environment.
2. Execute transitions:
    a. Execute transitions for uncontrollable events in an event loop, to update the state based on the inputs.
    b. Execute transitions for controllable events in an event loop, to compute the control

response, and update the state to reflect it.

3. The output variables are written to the environment.

Event loops

To ensure a consistent transition execution order, each event loop is executed in a specific manner:

- The events are ordered in some way (see event order below).
- For each event, in order, at most one transition is executed per iteration of the loop.
- If the event is enabled, one transition is executed for it. If the event is disabled, it is skipped.
- After each event, regardless of whether it was enabled and a transition was executed for it, the next event is considered, and so on for all the events of the particular loop.
- Once all events have been considered once, if for any of the events a transition was executed, then another iteration of the loop is performed.
- The loop is executed as many times as necessary, until no event is enabled anymore. However, a limit may be set and then the loop is executed as most as many times as the limit indicates. For instance, the limit may be derived from the result of the bounded response check.

To ensure consistent execution of transitions for events, the edges that participate must be chosen in a specific manner:

- If the event is a channel, the first enabled sender automaton for that channel participates in the transition (see automaton order below).
- If the event is a channel, the first enabled receiver automaton for that channel participates in the transition (see automaton order below).
- For each participating automaton for the event (sender, receiver, monitor or non-monitor synchronizing automaton), the first enabled edge participates in the transition (see location and edge orders below).
- For a monitor automaton for the event, if there is no enabled edge in the current location of the automaton, the automaton is ignored for that transition.

To ensure a consistent transition execution order, various elements of CIF specifications must be considered one-by-one in a specific order:

- The events of the specification are sorted (see sorting details below).
- The automata of the specification are sorted (see sorting details below).
- The location of an automaton must be considered in model order, the order in which they are defined in the automaton.
- The edges of a location must be considered in the model order, the order in which they are defined in the location.

Note that the order of the events on an edge, in case an edge has multiple events, does not affect the transition execution order.

For objects of CIF specification that are sorted, the sorting must be done in a specific manner:

- The absolute non-escaped names of the objects must be sorted.
- Sorting must be based on the ASCII characters of the names.
- The names must primarily be sorted case-insensitively, but in case they are the same, they must secondarily be sorted case-sensitively.
- The names must be sorted in ascending order, such that for instance `button` is placed before `motor`, and thus `button` is considered before `motor`.
- A prefix must be placed before any longer names of which it is a prefix, such that `button` is placed before `button2`.

Runtime errors

The model is assumed not to contain runtime errors, such as division by zero, modulus of zero, or assigning out of bounds values. Since the checker silently discards any such behavior, the outcome of the various checks may be incorrect if the model exhibits behavior leading to runtime errors.

When PLC code is generated for models with runtime errors, the properties cannot be guaranteed as the resulting code may not behave as the CIF specification. See also the section about supported specifications of the PLC code generator.

One way to get a model without runtime errors is to use supervisor synthesis, since that removes such behavior from its input. For more information, see the data-based and event-based supervisor synthesis tools. Alternatively, the CIF explorer may be used to verify that the specification does not contain runtime errors.

References

- [Reijnen et al. (2019)] Ferdie F.H. Reijnen, Albert T. Hofkamp, Joanna M. van de Mortel-Fronczak, Michel A. Reniers and Jacobus E. Rooda, "Finite Response and Confluence of State-based Supervisory Controllers", In: Proceedings of the 15th International Conference on Automation Science and Engineering, pages 509-516, 2019, doi:10.1109/COASE.2019.8843335

Bounded response check

The controller properties checker can check a supervisor for having bounded response.

The bounded response property holds if for each execution of the code, the number of iterations of the event loops to execute for uncontrollable and controllable events are both bounded. This property ensures that the code that executes the uncontrollable events, as well as the code that executes the controllable events, always terminate. That is, the system has no loops with only uncontrollable events, or with only controllable events. In other words, there is no livelock for uncontrollable events, nor for controllable events.

If bounded response doesn't hold, additional information is printed to the console:

- The edges that are part of a cycle. Note that there could be multiple cycles, and one edge could be part of more than one cycle.

- The states that are part of a cycle. Only states at that can be reached at the end of executions of the corresponding event loop are printed, not states that occur within the event loop during those executions. Hence, it could be that for some cycles only some of the states are printed. Note that states could be part of multiple cycles.

  Irrelevant variables are left out when states are printed, and therefore each *printed cycle state* could actually represent multiple *actual cycle states*. That is, an irrelevant variable that is left out can have any value according to its type, and the number of actual cycle states represented by the printed cycle state is therefore determined by the number of combinations of possible values for the variables that are left out. For instance, consider a model that has variables `a`, `b` and `c` of type `bool`. If `a=true` is printed, then this printed cycle state represents four actual cycle states:

  1. `a=true, b=false, c=false`
  2. `a=true, b=false, c=true`
  3. `a=true, b=true, c=false`
  4. `a=true, b=true, c=true`

  Since the number of printed cycle states can be very large, by default only 10 printed cycle states are printed to the console. The limit on the number of printed cycle states to print can be changed by configuring the *Maximum number of printed cycle states* option.

Examples

The check verifies lack of transition loops that consist of only uncontrollable or only controllable events. For example, the `Actuator` automaton in the example below has bounded response, because between transitions for events `c_on` and `c_off`, the location of `Sensor` has to change via an uncontrollable event. The `Sensor` automaton does not have bounded response, as the sensor can keep go on and off forever, without restrictions.

```
plant Actuator:
  controllable c_on, c_off;

  location Off:
    initial;
    edge c_on when Sensor.On goto On;
  location On:
    edge c_off when Sensor.Off goto Off;
end

plant Sensor:
  uncontrollable u_on, u_off;

  location Off:
    initial;
    edge u_on goto On;
  location On:
    edge u_off goto Off;
```

```
end
```

The model below also does not have bounded response. Whenever `StartButton` and `StopButton` are both equal to `true`, the motor keeps starting and stopping.

```
input bool StartButton, StopButton;

plant Motor:
  controllable c_on, c_off;

  location Off:
    initial;
    edge c_on when StartButton goto On;
  location On:
    edge c_off when StopButton goto Off;
end
```

Implementation details

The computation of bounded response takes into account the execution scheme. Bounded response for a model is computed as follows:

1. If there are no initial states, the bounds are both trivially zero.
2. Otherwise, we compute the possible start states for the uncontrollable and controllable event loops.
3. Then we compute for each event loop its bound, using its possible start states as input.

The possible start states that we can be in before the uncontrollable and controllable event loops are computed as follows:

1. We have three event loops, one for the input variable events (which can change before the 'program code' is executed), the uncontrollable events, and the controllable events. We initialize the states we can be in at the start of the input variables events loop to the initial states of the system. We initialize the states we can be in at the start of the uncontrollable and controllable event loops to *false*.
2. We execute as many rounds as needed to reach a fixed point for the three types of start states, by in each round executing the 'program code' once:
    a. In a round, we first do an event loop for the input variable events, starting from the currently known possible start states for that loop. This allows input variables to change value. The result of executing that loop are the new possible states that the uncontrollable events loop can start in.
    b. We then similarly perform an event loop for the uncontrollable events, from its known possible start states. We then update the start states of the controllable events loop based on the states we can be in at the end of the uncontrollable events loop.
    c. We then similarly perform an event loop for the controllable events, from its known possible start states. We then update the start states of the input variable events loop based

on the states we can be in at the end of the controllable events loop.

The end states we can be in after executing a single full event loop from given start states, are computed as follows:

- The algorithm is similar to that of computing the bound (see below), but with a small difference. Besides tracking which states we can currently be in, at each point of the execution, we also track the states we can be in at the ends of different iterations. This is required due the possibility of having cycles in the event sequences of the specification.

  If for example in one iteration a variable *x* is inverted from *true* to *false*, and in the next iteration it is inverted again from *false* to *true*, then each iteration ends in different states than the previous one. We thus can't determine a fixed point that way.

  This is not a problem when computing the bound, as then we know that every state in the cycle is present, and thus they all go to the next state in the cycle, and we still get a fixed point. But here, we are computing the loop start states, so we don't have them yet, and we thus can't rely on them yet.

  Here, we therefore compute all possible end states of all iterations of the loop, and detect a fixed point for that. This fixed point will be reached, since we can't forever reach new end states for next iterations.

  Note that we use the 'all iterations end states' to detect the fixed point, but still we return only the end states we reached after the last iteration as the result of this computation.

The end states we can be in after executing a single iteration of an event loop from given start states, are computed as follows:

- To match the execution scheme, we try each event in the prescribed order, and execute at most one transition for it per iteration.
- The states in which an event can start, are:
  - For the first event: the given start states.
  - For all other events: the states in which the previous event could end.
- To take an edge for an event, we consider the states we can be in before taking the edge. We compute the states in which we can end up by taking the edge. We also consider the states we can remain in, if the guard of the edge is not enabled. We combine them, to get the states after possibly taking a transition for the edge. We then continue with the next edge for the same event, but only from the states in which the previous edges for that same event were not enabled. The result of possibly applying at most one transition for the event is the combination of all the possible target states in which we can end up by applying the different edges (for different source states), and the states in which we may remain because none of the guards of the edges for the event is enabled.

The bound for an event loop, given the possible start states for that event loop, is computed as follows:

- We start with all states we can be in before the event loop, to account for execution starting in

any of those states.

- If there are no events, the bounds are trivially zero. Otherwise, the current states are those states where we can be after applying zero transitions from the start of the execution (bound zero).
- We find the bound by in each iteration taking at most one transition for the relevant events. In each iteration, we thus try to take a transition per event, and we end up with the states we can be in after that iteration.
- If the specification has bounded response, all sequences are of finite length, and at some point they end. That is, after enough iterations, given the length of the sequence and the transition execution order, no more transitions can be executed for the events. If in iteration *n* no more transitions can be executed, then the possible states we can be in before and after that iteration are the same, and we have reached a fixed point. Since no transitions were executed in iteration *n*, *n - 1* iterations is then enough, and the bound is thus *n - 1*.
- If the specification does not have bounded response (assuming the execution scheme), then there is at least one cycle of states. The states within a cycle will with one transition go to the next state in their cycle. Cycles are thus preserved with each iteration. If all bounded sequences at some point end, then only the cycles (there could be more of them) remain. Then a fixed point is reached, but since transitions can still be executed, there is no bounded response.

Confluence

The controller properties checker can check a supervisor for having confluence.

The confluence property holds if for each execution of the controllable events, it does not matter in which order the controllable events are executed, as the same state is reached regardless. This property ensures that the controller can make any choice between controllable events without losing the guarantees of synthesis. By extension, this means that code generators may generate code for the controller that executes the controllable events in any order.

This check does not check for confluence of the uncontrollable events. Furthermore, the check may produce false negatives.

If confluence may not hold, the event pairs for which confluence could not be decided are printed to the console.

Examples

The check verifies that for every state in the model with several edges enabled at the same time, any taken edge will lead to the same decision. The `Actuator` automaton below has two different edges with controllable events from the same location, but only one of them can be enabled at the same time since the `Sensor` automaton cannot be in two locations at the same time. As a result, the automaton does not need to choose between edges and the model is confluent.

```
plant Actuator:
  controllable events c_a, c_b;
  disc bool on = false;
```

```
  location A:
    initial;
    edge c_a when Sensor.On do on := true;
    edge c_b when Sensor.Off do on := false;
end

plant Sensor:
  uncontrollable u_on, u_off;

  location Off:
    initial;
    edge u_on goto On;
  location On:
    edge u_off goto Off;
end
```

Note that the confluence property can only hold for models with finite response. Only if a controller determines it control responses in a finite number of steps, does it matter whether the control response is then always the same.

Implementation details

Confluence is determined by checking that for all edge pairs with different controllable events, one of the following cases holds:

Mutual exclusiveness

If the edge with the first event is never enabled at the same time as the edge with the second event, this case applies. Only one of the edges can be taken at any time, there is never a choice between the edges.

```
edge a when x = 10 goto ...;

edge b when x = 12 goto ...;
```

The guard holds for at most one of the edges, and thus it is impossible to pick a 'wrong' edge. This case also applies if the edges start from different locations in the same automaton.

Update equivalence

The edge with the first event is enabled at the same time as the edge with the second event, but they both change the system state in the same way.

```
edge a do x := 5 goto P;

edge b do x := 5 goto P;
```

As either event has the same effect, it does not matter which edge is taken.

Independence

The edge with the first event leads to a state with an edge with the second event and vice versa. Both sequences make the same changes.

```
  edge a do x := 5 goto P;

  edge b do y := 7 goto Q;

location P:
  edge b do y := 7 goto R;
location Q:
  edge a do x := 5 goto R;
```

While first taking the `a` edge leads to a different state than first taking the `b` edge, they converge again after the next edge.

Skippable

The edge with the second event may be taken, but its change is overridden.

```
  edge a do x := 6 goto Q;

  edge b do x := 1 goto P;

location P:
  edge a do x := 6 goto Q;
```

The edge with event `b` temporarily changes `x` and jumps to a different location, but there an edge with the `a` event can be taken and it overwrites all changes made by the `b` edge.

Reversible

A pair of unrelated events may or may not occur.

```
  location P:
    edge a goto Q;

automaton Z:
  location A:
    edge b when P goto B;
  location B:
    edge c when Q goto A;
```

Here the choice between events `a` and `b` matters, since taking the edge with event `a` disables event `b`. However, if the edge with event `b` is taken first, then after event `a`, a third event `c` reverts the change made by the edge with event `b`. Therefore it does not matter whether events `b` and `c` are taken, since they cancel each other out.

If a match can be found for all pairs, the checker concludes that the confluence property holds. Otherwise, it indicates that a proof of the confluence property could not be found. As the check is an over-approximation, there may be false negatives (the property holds but that could not be proven due to limitations in the checks).

This algorithm is based on [Reijnen et al. (2019)].

False negatives

The confluence check is an over-approximation of edges for controllable event pairs. As a result, the check might indicate that the specification may not be confluent, while in reality it is confluent. When the check indicates that there is confluence, this is always correct.

False negatives may be reported in the following situations:

- Checked edge pairs may be in the non-reachable part of the state space.
- The system continues to behave in the same way after diverging.
- Not all edges of an event pair match with the same case.
- Guard conditions or assigned values have no effect if you take the reachable state space into account.

By using the CIF explorer before performing the check some of the above causes are eliminated.

Note that besides false negatives, the result of this check may be incorrect for models with runtime errors.

Finite response check

The controller properties checker can check a supervisor for having finite response.

The finite response property holds if for each execution of the code, the number of transitions to execute for controllable events is finite. This property ensures that the code that executes the controllable events always terminates. That is, the system has no loops with only controllable events. In other words, there is no livelock for controllable events.

This check does not check for finite response of the uncontrollable events. Furthermore, the check may produce false negatives. Hence, it is recommended to use the bounded response check instead.

If finite response may not hold, the events that might still occur in a controllable-event loop are printed to the console. This can be disabled using an option.

Examples

The check verifies lack of event loops that consist of only controllable events. For example, the `Actuator` automaton in the example below has finite response, because between event `c_on` and `c_off`, the location of `Sensor` has to change via an uncontrollable event.

```
plant Actuator:
  controllable c_on, c_off;
```

```
  location Off:
    initial;
    edge c_on when Sensor.On goto On;
  location On:
    edge c_off when Sensor.Off goto Off;
end

plant Sensor:
  uncontrollable u_on, u_off;

  location Off:
    initial;
    edge u_on goto On;
  location On:
    edge u_off goto Off;
end
```

The model below does not have finite response. Whenever `StartButton` and `StopButton` are both equal to `true`, the motor keeps starting and stopping.

```
input bool StartButton, StopButton;

plant Motor:
  controllable c_on, c_off;

  location Off:
    initial;
    edge c_on when StartButton goto On;
  location On:
    edge c_off when StopButton goto Off;
end
```

Implementation details

Finite response for a model is determined as follows:

1. Find controllable-event loops in automata. For this, guards and updates are omitted, also see false negatives.
2. Find variables that are never updated by controllable events. This includes by definition all input variables.
3. For all events in a controllable-event loop, determine whether the guards are mutually exclusive. For this, only the variables found in step 2 are considered. For the guards, all other edge guards and state/event exclusion conditions are included.
4. If there are events in the alphabet of an automaton, but not in any of its controllable-event loops, this event is removed from the set of controllable events.
5. If the set of controllable events changed, repeat the process.

6. If the set of controllable events is empty, the model has finite response. Otherwise, it can not be concluded there is finite response. Note that the check is an over-approximation, and there may be false negatives.

This algorithm is based on [Reijnen et al. (2019)].

False negatives

The finite response check is an over-approximation of the existence of controllable event-loops. As a result, the check might indicate that the specification may not have finite response, while in reality it has finite response. When the check indicates that there is finite response, this is always correct. The bounded response check does not suffer from false negatives, and is recommended to be used instead of the finite response check.

False negatives may be reported in the following situations: * Loops exist in the non-reachable part of the statespace. * Edges contain guards or updates.

While determining loops in the automata, only explicit loops are considered. That is, the algorithm omits guards and updates of the variables. For that reason, in the example below, `c_on` and `c_off` are two independent loops (instead of `c_on, c_off` if the guards and update were included).

```
plant Actuator:
  controllable c_on, c_off;
  disc bool on = false;
  location:
    initial;
    edge c_on  when not on do on := true;
    edge c_off when     on do on := false;
end
```

To reduce the number of false negatives, it is advised to use locations instead of variables whenever possible. False negatives may also be avoided by using the CIF explorer on the input specification before performing the check.

Note that besides false negatives, the result of this check may be incorrect for models with runtime errors.

Non-blocking under control check

The controller properties checker can check a supervisor for being non-blocking under control.

The non-blocking under control property holds if a marked state can be reached, with the transitions for uncontrollable events and controllable events executed separately in code generated from the supervisor. This property ensures that the code is still non-blocking, and that the non-blockingness guarantee of synthesis is thus preserved in generated code.

Concretely, the check verifies that for every reachable state, there exists a controllable-complete path. That is, for every reachable state, there exists a path to a marked state, with no controllable events enabled in the last/marked state of the path, and all transitions on the path are either for

controllable events, or for uncontrollable ones from states where no controllable events are enabled.

Examples

Consider the following example model, with a Supervisor1a automaton. The non-blocking under control property holds for this model.

All locations are reachable. From locations Start, Done1 and Done2, controllable event sequences are possible to marked location BothDone, where no controllable event is possible. There is thus a marked location at the end of the controllable-event paths consisting of c_act1 and c_act2 events. In location BothDone, u_button_pushed is enabled and no controllable events can be taken. From that location, u_button_pushed can thus be taken, followed by the controllable events.

Note that event u_button_pushed is monitored, which means the event is also possible in the other locations. That is ok, since a controllable-complete path must exist, and does for every reachable state. That other uncontrollable events are enabled in locations where controllable events are taken on the controllable-event path, does not invalidate the property. The property only puts requirements on uncontrollable events that are taken as part of a controllable-complete path, as they may only occur from states where no controllable events are enabled.

```
supervisor Supervisor1a:
  controllable c_act1, c_act2;
  uncontrollable u_button_pushed;
  monitor u_button_pushed;

  location Start:
    initial;
    edge c_act1 goto Done1;
    edge c_act2 goto Done2;

  location Done1:
    edge c_act2 goto BothDone;

  location Done2:
    edge c_act1 goto BothDone;

  location BothDone:
    marked;
    edge u_button_pushed goto Start;
end
```

Now consider a slightly different version of this model, as shown below. Automaton Supervisor1b is similar to Supervisor1a, but has a different marked location. For Supervisor1b, the non-blocking under control property does *not* hold.

All locations are still reachable. And if c_act1 is done first, before c_act2, the automaton goes through marked location Done1. But, if instead c_act2 is done first, before c_act1, the automaton does not go through a marked location. And if every time the choice is made to first do c_act2

before c_act1, then the automaton never goes through a marked location. The automaton is can thus exhibit blocking behavior, depending on the order in which the events are executed in generated code.

The non-blocking under control property does not hold here, since in the only marked location there is a controllable event enabled. The marked location should be at the end of the controllable-events path, not within it.

```
supervisor Supervisor1b:
  controllable c_act1, c_act2;
  uncontrollable u_button_pushed;
  monitor u_button_pushed;

  location Start:
    initial;
    edge c_act1 goto Done1;
    edge c_act2 goto Done2;

  location Done1:
    marked;
    edge c_act2 goto BothDone;

  location Done2:
    edge c_act1 goto BothDone;

  location BothDone:
    edge u_button_pushed goto Start;
end
```

As another example, consider the Supervisor2a automaton below. This model is a simplified version of Supervisor1a, where c_act1 is always done first, before c_act2. For this model, the non-blocking under control property also holds.

It still has the marked location at the end of the controllable-event path. Also, the uncontrollable event u_button_pushed is still possible from that marked location, where no controllable event is enabled.

```
supervisor Supervisor2a:
  controllable c_act1, c_act2;
  uncontrollable u_button_pushed;
  monitor u_button_pushed;

  location Start:
    initial;
    edge c_act1 goto Done1;

  location Done1:
    edge c_act2 goto BothDone;
```

```
  location BothDone:
    marked;
    edge u_button_pushed goto Start;
end
```

Now consider a slightly different version of this model, as shown below. Compared to `Supervisor2a`, automaton `Supervisor2b` has the additional option to do `u_button_pushed` between `c_act1` and `c_act2`. And only by this additional edge, can it reach a marked location. For `Supervisor2b`, the non-blocking under control property does *not* hold.

Still, all locations are reachable. But, if first `c_act1` is done, then `c_act2`, and finally `u_button_pusehd`, then it cycles without going through a marked location. A marked location can of course be reached, by doing `u_button_pushed` in location `Done1`. But, in the generated code first all transitions for uncontrollable events are executed, and then the transitions for controllable events. In `Start`, no uncontrollable event is possible. And after doing `c_act1` the code will keep doing transitions for controllable events, opting for `c_act2` rather than `u_button_pushed`. Hence, a marked state can never be reached in the generated code. A marked state can only be reached by doing the uncontrollable event in between controllable events, which is excluded by the execution model of the generated code.

A controllable-complete path may only include an uncontrollable event if from its source state no controllable events are enabled. Uncontrollable event `u_button_pushed` may thus not be included from location `Done1`, as there `c_act2` is enabled. The controllable-complete path can thus never reach the marked location. The non-blocking under control property therefore doesn't hold in this model.

```
supervisor Supervisor2b:
  controllable c_act1, c_act2;
  uncontrollable u_button_pushed;
  monitor u_button_pushed;

  location Start:
    initial;
    edge c_act1 goto Done1;

  location Done1:
    edge c_act2 goto BothDone;
    edge u_button_pushed goto Halt;

  location BothDone:
    edge u_button_pushed goto Start;

  location Halt:
    marked;
end
```

Implementation details

Non-blocking under control for a model is determined as follows:

- Compute predicate *gc* that indicates in which states a controllable event is enabled in the system. This is computed by taking the disjunction of the guards of the edges labeled with controllable events.
- Compute predicate *not gc* that indicates in which states no controllable events are enabled in the system. This is computed by taking the inverse of *gc*.
- Compute the *ccp* states, the states on controllable-complete paths. This is computed by performing a backwards reachability computation from *marked and not gc* states. During this reachability computation, for each edge labeled with an uncontrollable event, including edges for allowing input variables to change value, *guard and not gc* is used instead of its original *guard*.
- Compute the *bad* states, the not-*ccp* states and states that can reach such states. This is computed by performing a backwards reachability computation from *not ccp* states. Unlike in the previous step, the original guards are used for all edges during this reachability computation.
- The model is non-blocking under control if the *initial* states are not *bad*. That is, the model is non-blocking under control check if *(initial and bad) = false* holds.

This algorithm is based on [Reijnen et al. (2019)].

4.5. Real-time testing, code generation, and implementation tools

4.5.1. CIF code generator

The CIF code generator can be used to generate implementation code for various languages/platforms. The generated code is complete, and can be used as is, without any CIF related library or runtime.

The remainder of this page explains general information about the tool, that applies to all target languages/platforms. Specific information is available for each of the target languages/platforms, on separate pages:

- C89 (if there is no `C99` compiler)
- C99
- Java
- JavaScript
- HTML
- Simulink, with an example of controlling a tank

The generated code adheres to the execution scheme defined by the CIF controller properties

checker.

Starting the generator

The generator can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF implementation tools › Generate code for CIF specification...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF implementation tools › Generate code for CIF specification...**.
- Use the `cifcodegen` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifcodegen` command line tool.

Options

Besides the general application options, this application has the following options that apply to all target languages/platforms:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output directory*: The absolute or relative local file system path to the output directory. This is the directory to which the generated code files will be written. If the directory does not yet exist, it is automatically created. By default, the current directory is used.
- *Target language*: Specifies the target language/platform for which to generate code. By default, HTML is generated. The following languages/platforms are supported:

Target language/platform	Command line value
C89	`c89`
C99	`c99`
Java	`java`
JavaScript	`javascript`
HTML	`html`
Simulink	`simulink`

  The first column indicates the target language/platform, and is linked to a page with more information for that specific language/platform. The second column indicates the command line value to use for the option, to select that target language/platform.

- *Code prefix*: The code prefix, used to prefix file names, identifiers in the code, etc. The prefix must be a valid CIF identifier, i.e. consist of only letters (`a` to `z`, `A` to `Z`), numbers (`0` to `9`), and underscores (`_`), and not start with a number. If no prefix is given, the prefix is derived from the

input filename, by removing the `.cif` file extension (if present), and adapting the result to be a valid CIF identifier. This is also the default.

Supported specifications

The CIF code generator supports a subset of CIF specifications. The following restrictions apply:

- Specifications without automata are not supported.
- Only events that are declared as controllable or uncontrollable are supported. All other events are not supported, including the `tau` event when used explicitly or as an edge without events.
- Initialization predicates in components are not supported, except if it can be determined statically that they are trivially `true`.
- State invariants (in components as well as locations) are not supported, except if it can be determined statically that they are trivially `true`.
- Discrete variables with multiple initial values (including `any`) are not supported.
- External user-defined functions are not supported.
- Urgent locations are not supported.
- Initialization predicates in locations that can not be statically evaluated are not supported.
- Automata that do not have exactly one initial location are not supported.
- Urgent edges are not supported.
- Data types other than `bool`, `int` (with or without range), `real`, `string`, enumerations, tuples, and arrays, are not supported. This applies to the data types of variables, parameters of functions, return types of functions, etc.
- Sampling of distributions is not supported.
- The use of the conjunction and disjunction binary operators on anything other than boolean values is not supported.
- The use of the equality and inequality binary operators on anything other than boolean, integer, real, string, or enumeration values is not supported.
- The use of the addition binary operators on anything other than numeric, string or array values is not supported.
- The use of the subtraction binary operators on anything other than numeric values is not supported.
- The use of the element test and subset binary operators is not supported.
- Projection on anything other than tuples, arrays, and strings is not supported. This applies to expressions as well as addressables (the left hand sides of assignments). For arrays, both 0-based indices (counting from the left) as well as negative indices (counting from the right) are supported.
- Slicing is not supported.
- Function calls on anything other than standard library functions and internal user-defined

functions is not supported.

- The `del`, `pop`, `acosh`, `asinh`, `atanh`, `cosh`, `sinh`, and `tanh` standard library functions are not supported.
- The `empty` standard library function on anything other than arrays is not supported.
- The `size` standard library function on anything other than strings and arrays is not supported.
- The distribution standard library functions are not supported.
- Lists (except for arrays), sets, and dictionaries are not supported.
- The use of functions as values is not supported. That is, functions may only be used in function calls, and may for instance not be stored in variables, or passed to other functions.
- Print declarations with pre/source state text and post/target state filtering (`when`) are not supported.
- Print declarations with post/target state text and pre/source state filtering (`when`) are not supported.
- CIF/SVG declarations are supported only for HTML code generation. If the specification connects to an SVG image, but no CIF/SVG input or output mappings apply to that image, a warning is printed. For the other target languages, a warning is printed if any CIF/SVG declarations are encountered, and these declarations are then ignored.

The restrictions on types and expressions do not apply to their use in values of documentation annotation arguments.

Specific target languages may add additional restrictions. Check their specific information for the details.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove CIF/SVG declarations (if the target language is not HTML)
- Remove annotations (all annotations except documentation annotations are removed)
- Eliminate component definition/instantiation
- Eliminate state/event exclusion invariants
- Simplify values (no references, optimized)
- Simplify others

After these preprocessing steps, the generator checks whether the specification is supported. It then applies the following additional CIF to CIF transformations:

- Linearize (merge)
- Eliminate type declarations

- Push print file declarations into print declarations
- Push SVG file declarations into other CIF/SVG declarations
- Merge enumerations
- Simplify values (no references, optimized)
- Add default initial values

Generated code

This section explains the basics behind the generated code, as they apply to all target languages. The specific pages that explain the code generated for specific target languages provide further details.

For all languages, the generated code adheres to the execution scheme defined by the CIF controller properties checker. Note however that CIF specifications with SVG input mappings that map to events should be avoided if full compliance is desired, and SVG input mappings with updates to input variables should be used instead.

Code overview

The code first ensures that input variables have a value. The values of the input variables are obtained from the environment (for all targets except HTML), or initialized to their default values (for the HTML target, only for the first execution). Then, the initial state is initialized (for the first execution only), and the values of the continuous variables are updated to account for the amount of time that has passed (for all but the first execution). Then transitions executed (see below). Finally, output values may be supplied to the environment.

Code that executes transitions of a CIF model is called *event loop* code. The generated code for a CIF specification has multiple event loops. If the target supports SVG input mappings, the first event loop is for the environment events for SVG input mappings with updates, and for the interactive SVG input events resulting from SVG input mappings with events. For all targets, what follows is an event loop for the edges with uncontrollable events, and one for the edges with controllable events.

For further details, see the execution scheme defined by the CIF controller properties checker. Note that SVG input mappings that map to events are not strictly compatible to the scheme. If strict adherence is desired, SVG input mappings with updates should be used instead.

Repeated execution

During the execution of the code, variable `time` from the CIF model is not updated. Execution of the code takes time. After the code is executed, potentially more time passes. Then, the code can be executed again. The time (in seconds) between the first execution and the current execution is the total time that has passed since execution started, and this becomes the new value of variable `time`.

How often the code is executed, and whether this is done with a fixed cycle time (periodic task scheduling) or with a variable cycle times (non-periodic tasks) can be influenced by the end user. The details differ per target language, and can be found on their respective pages.

Event loops

The CIF model is first linearized to eliminate parallel composition. The result is self loop edges with guards and updates. The main part of the generated code essentially consists of update code that is conditionally executed. This code is part of the event loops, that execute edges for the various events (and SVG input mappings, if the target is HTML and the model contains SVG input mappings). Each event loop repeatedly executes its enabled edges, until no more edges are enabled. Once no more edges are enabled, an event loop terminates. The order in which the code is generated for the different edges (in the *event loops*), is done in such a way that it matches the execution scheme defined by the CIF controller properties checker.

Termination

If in the CIF model always at least one event is enabled, the generated code will run forever, and will never terminate. Time will not pass. Code generation should be avoided for such models. It is the responsibility of the modeler to ensure this is taken into account. A bounded response check can be done to ensure the CIF specification does not have infinite sequences of enabled events.

Predictability of execution time

If during an iteration of an event loop none of its edges is enabled, the loop terminates. If at least one of its edges is enabled, the event loop is executed again.

This approach generally does not result in predictable execution times of the program body. That is, if a certain edge keeps being enabled, its loop is executed over and over again. In that case, the execution time of the body may exceed the fixed cycle time, and the controller is no longer guaranteed to work correctly.

Even if a loop is only executed a finite number of times, the execution time of the body may exceed the fixed cycle time. One may consider monitoring the execution times to detect such issues. All implementations provide a way to obtain execution time statistics. The details differ per target language, and can be found on their respective pages.

Optional execution feedback

Several implementations provide additional information during execution, such as which events have been executed, timing information, and what new print output has been generated. Such information can generally be enabled and disabled in the implementations. The details differ per target language, and can be found on their respective pages.

Correct handling of I/O

The generated code cannot guarantee its correct behavior if the inputs from the external I/O change more rapidly than the generated code is executed. That is, if an input changes during a single execution of the code, the change is not noticed until the next execution. If an input changes during an execution and changes back during that same execution, the change is not noticed at all. Therefore, executing the code often enough per second, is essential. In other words, choosing an appropriate cycle time is essential for the correct functioning of the generated code.

Accuracy over time

The generated C89, C99, Java, JavaScript and HTML code maintains the running time in seconds. Due to the finite representation of numeric values, this results in numeric overflow after execution for a long time. Running the code continuously for longer than 100 years should be avoided.

The continuous variables from the CIF specification in the generated code are updated each execution using the Euler method for integration. That is, each execution the time that has elapsed since the previous execution is multiplied by the derivative of the continuous variable in the state of the previous execution. This value is added to the value of the continuous variable. Essentially, a linear approximation of the derivative is used, calculated using the state of the previous execution.

The Euler method can be numerically unstable, it can suffer from rounding errors due to the use finite binary representations of real values, and has some other issues as well. The effects are likely to be limited if linear continuous variable (continuous variables with constant derivatives) are used. Especially clocks (derivative +1 or -1) generally don't suffer too much from these issues. However, even clocks may suffer from loss of accuracy over time. As such, clocks should not be used to measure over long periods of time. Instead they should be reset, rendering the past loss of accuracy irrelevant for any future measurements.

Enumerations

One of the preprocessing transformations merges all enumerations into a single enumeration. This is done in such a way that the behavior of the original CIF model is left intact. However, it is up to the implementer that connects the inputs from the environment to the generated code, to ensure that only valid enumeration literals are assigned to the input variables of the generated code.

Annotations

All annotations in the specification are ignored, except for documentation annotations:

- The documentation of algebraic parameters, algebraic variables, constants, discrete variables, enumeration literals, events, function parameters, function variables, functions, input variables, locations of automata with at least two locations, and specifications is included in the generated code.
- The documentation of continuous variables is also included in the generated code, except when generating Simulink code.
- The documentation of automata, automaton definitions, automaton instantiations, edges, enumerations, groups, group definitions, group instantiations, invariants, locations of automata with only one location, and type declarations is ignored.

C89 code generation

The CIF code generator can generate C89 code from a CIF specification. It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to C89 code generation.

In general, use of the C99 code generator is preferred, as the language contains a decade longer development. However, in the case there is no `c99` compiler available, the `c89` code generation might be a useful fallback option to convert a CIF program for controlling a real time system. The

conversion uses a simple integrator intended for updating running clocks (derivative should not change during a time step for best results). Also, it does not compute length of the next time step, but instead assumes being called regularly to update internal state and perform discrete steps if possible.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

C89 code generation has the following additional restrictions:

- List comprehension expressions are not supported.

Differences in output

When converting a real number to text, for example when printing a real value, the generated C89 code does not preserve trailing zeroes of real number fractions while using `%g`. For example `fmt("%.3g", 1.0)` will output `1` rather than `1.000`.

This may cause real number values to be printed with less digits precision than indicated in the CIF program.

Strings have an upper limit in length that is set during compilation. Longer strings are silently truncated to this length.

Options

The C89 code generator only uses the common options that apply to all target languages/platforms of the CIF code generator, it has no options of its own.

The code prefix that can be configured using a common option is used to prefix external names, making them unique for the generated code.

Generated files

C89 code generation leads to creation of seven files:

Generated file	Purpose
`<prefix>_library.h`	Runtime support code and library headers
`<prefix>_library.c`	Runtime support code and library implementation
`<prefix>_engine.h`	Interface definition of the translated CIF program
`<prefix>_engine.c`	Implementation of the translated CIF program
`<prefix>_compile.sh`	Compile script for Linux
`<prefix>_test_code.c`	Example of external functions, allow compiling the generated code
`<prefix>_readme.txt`	Short description of use, the interface, and compile options

where `<prefix>` is replaced by the value of the *Code prefix* option during code generation.

The aim is to provide a complete and compilable package, to minimize the effort in converting a CIF specification. Most files are static text (modulo the `<prefix>` replacement). The two exceptions are both engine files that contain the C89 version of the CIF specification. The interface of these files is however fixed (modulo the `<prefix>` replacement), and generated data types and operations on them have predictable names, reducing the chance of failed compilation when re-generating the files.

When modifying a generated file (such as the compile script or the example test code), keep in mind that running the code generator again will overwrite *all* files. Rename a file before modifying it to avoid loss of work.

Compilation of the generated code

The generated code is C89 code that can be compiled using a C89 C compiler, like `gcc`:

```
gcc -Wall -std=c89 -DPRINT_OUTPUT=1 -DCHECK_RANGES=1 \
    <prefix>_engine.c <prefix>_library.c <prefix>_test_code.c -lm
```

The single `\` is not part of the command, it means the next line should be appended at the end.

The `<prefix>` should be replaced by the value of the *Code prefix* option during code generation. The `PRINT_OUTPUT` and `CHECK_RANGES` macros enable support for the CIF `print` statement and checking of integer range overlap, as defined by CIF. In general however, you may want to replace the test code file by your own code.

C89 compile-time options

Besides the `PRINT_OUTPUT` and `CHECK_RANGES` macros, the generated code has a few other compile-time options to customize its behavior. Below is the complete list.

Macro name	Effect
`MAX_STRING_SIZE`	CIF has variable length strings, in the generated code they are converted to strings with a fixed upper limit. Default is 128 (including the terminator character). Set this macro for a different limit. *Note that both the library and the engine use this macro. It is crucial that both files use the same value for this macro.*
`CHECK_RANGES`	When copying integers from one range to another, where the latter is only partially covered by the former, CIF checks whether the copied values do not violate the destination range. Since violating this property generally indicates a programming error, it is recommended to enable this macro.
`EVENT_OUTPUT`	If set, the generated code calls `<prefix>_InfoEvent()` before and after performing an event to allow third-party code to act on it. Enable if your code needs to know when an event is performed.

Macro name	Effect
`PRINT_OUTPUT`	If set, the CIF `print` statement is performed, and the `<prefix>__PrintOutput()` is called with the resulting text and destination filename. Mainly intended for debugging, but only useful in systems where the text can be displayed to a user.
`MAX_NUM_EVENTS`	The CIF semantics state that execution of edges is instantaneous and time-less. The generated code loops to perform edges, until no edge can be executed any more. Unfortunately, such a loop is not time-less. To avoid requiring too much time, there is an upper limit of 1000 iterations. Set this macro to change the limit.
`KEEP_RUNNING`	The CIF language checks range conversions (if `CHECK_RANGES` is set) and verifies updates of continuous variable updates. Setting this flag disables a few checks. Most of these checks are in `assert()` checks and not controlled by this macro, since violating the condition generally means a crash is eventually inevitable. It is recommended to avoid using this macro.

Executing the code

The engine files act as a library that implements the CIF program. To run the program, the code must be called regularly. The code has two entry points, one for initializing and performing the first steps, and one for handling a time delay.

The first entry point is:

```
<prefix>_EngineFirstStep(void)
```

This entry point initializes all the data, queries the values of the input variables if present, and performs execution of edges until blocked on the first time step or until hitting the `MAX_NUM_EVENTS` limit. Since this resets the entire CIF program to its initial state, you should only call this when the system being controlled is also (re-)initialized.

The second entry point is more regularly called, after a period of time has passed:

```
<prefix>_EngineTimeStep(double delta)
```

By calling this entry point, you indicate that `delta` (> 0) time units have passed. The code reads new values of the input variables, updates the continuous variables, and performs edges until all edges are blocked, or hitting the `MAX_NUM_EVENTS` limit.

When the call of either entry point returns, you can query the value of the variables to use as input for other parts of the system.

Environment interface

Calling one of the above entry points causes the CIF program to be executed, up to the next time step (assuming the `MAX_NUM_EVENTS` limit is high enough). During the execution, the code performs a

number of callbacks to get information from the environment, to provide information about its actions, or to deliver output. In addition, after each call, variables may be inspected by the environment to get information about the decisions of the CIF program.

The externally provided callback functions that are being used are:

```
<prefix>_AssignInputVariables(void)
```

The CIF program requests new values for all its input variables. In your implementation of this function you should write values directly in the input variables. Input variables are assumed to be independent of other parts of the CIF program, in general you should not need other CIF program values to implement this function.

If `EVENT_OUTPUT` is set during compilation, executed events are reported with:

```
<prefix>_InfoEvent(<prefix>_Event_ event, BoolType pre)
```

If the `EVENT_OUTPUT` macro is set during compilation, the generated code reports with this call that event is about to (`pre` is `TRUE`), or has been (`pre` is `FALSE`) executed. In the pre-event call, the event can be executed, and all variables have their value before performing the updates of the involved edges. In the post-event call, all variables have their values updated as indicated in the updates of the edges.

Primary use of this call is to forward the decision to other parts of the controlled system.

Finally, if the `PRINT_OUTPUT` macro is set during compilation, the generated code reports lines of output to be printed with the following function call:

```
<prefix>_PrintOutput(const char *line, const char *fname)
```

It denotes that text `line` should be printed at file `fname`. The primary uses of this call are to enable debugging the system, or to log relevant events. Note that `fname` here can mean anything. It can mean a real file, but also a pseudo-device like `:stdout` which is commonly used to print output to the C `stdout` stream.

Data access

Each CIF constant is available as a variable that does not change after initialization. Each algebraic variable can be queried by calling a function with the same name as the variable. The value of continuous variables is available by accessing the equivalent variable in the generated code. The derivative of each continuous variable is available as function named by the variable, and the suffix `deriv`. The model time is available as `model_time`. There is no derivative function of time. Internal functions are available as well if required.

The engine header file lists the translated names. It should be easy to find the relevant entry, as the generated code also documents the original name.

Data types

The following table defines how CIF data types are converted in the C89 generated code.

CIF data type	C89 data type
`bool`	`BoolType` which is a typedef to `int`
`int`	`IntType` which is a typedef to `int`
`enum e = ...`	`<prefix>Enum` which is a typedef to `enum Enum<prefix>_`
`real`	`RealType` which is a typedef to `double`
`string`	`StringType` which is a typedef to a `struct`
`list[n] t`	`...Type` which is a typedef to a `struct`
`tuple(...)`	`...Type` which is a typedef to a `struct`

A few notes:

- The `...` in the C89 `...Type` name for arrays (fixed length lists) and tuples is the systematic type name of the array or tuple type.
- Due to preprocessing all enumerations are merged into one. The generated C89 code always has exactly one translated `enum`.
- The elementary data types (`BoolType`, `IntType`, `RealType`, and `StringType`) are defined in the library header file.
- The `StringType` structure wraps a `char data[MAX_STRING_SIZE]` array. The `struct` allows copying arrays by value using an assignment statement.
- The array type (list with a fixed length) structure wraps a `<element-type> data[<size>]` array for the same reason. Copying an array by assignment is allowed.
- The tuple type is a structure with fields `_field0`, `_field1`, and so on, where the type of each field in the structure matches with the type of the CIF tuple field.
- Boolean constants `true` and `false` are named `TRUE` and `FALSE` respectively.
- The `-0.0` value of doubles gets replaced by `0.0` to void subtle equality problems.
- Strings get silently truncated to `MAX_STRING_SIZE - 1` characters. They are always terminated with a NUL character.
- Enumeration values get `_<prefix>_` added in front to make them unique.

Systematic type names

For tuple and list types, the type is converted to a name in a systematic way, making each type unique and predictable.

The conversion composes a name of a type from the type of its elements, and concatenates the result. The following conversion table applies:

CIF type	Systematic name
`bool`	`B`
`int`	`I`
`enum e = ...`	`E` (Due to linearization, all enumerations are merged together to a single enumeration. As such, always exactly one `enum` is generated for a CIF model.)
`real`	`R`
`list[n] t`	`A<n><t>`, with `<n>` the length of the array, and `<t>` the systematic type name of the element type of the array.
`tuple(t1, t2, t3, ..., tn)`	`T<n><t1><t2><t3>...<tn>`, with `<n>` the number of fields in the tuple, and `<t1>` to `<tn>` the systematic type names of the fields.

For example, type `list[5] int` is converted to `A5I`, and type `tuple(list[5] int x; tuple(real a, b) z)` is converted to `T2A5IT2RR` (concatenation of `T2` with the field type names `A5I` and `T2RR`).

Runtime errors

The CIF language defines strict sets of allowed values for all data types, and performs checking of these values at run time. The generated C89 code follows that idea, and checks whether operations on data are safe. Most of these checks are performed with an `assert`. Executed C code does not provide useful stack traces by default, and original line numbers of e.g. a CIF program have no meaning due to the linearization that is performed before code is generated. In addition, the generated code is quite readable and it is not too hard to understand what the code is doing at some C source line.

C99 code generation

The CIF code generator can generate C99 code from a CIF specification. It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to C99 code generation.

The aim of the C99 code generation tool is to provide an easy conversion from a CIF program to control of a real time system. It uses a simple integrator intended for updating running clocks (derivative should not change during a time step for best results). Also, it does not compute length of the next time step, but instead assumes being called regularly to update internal state and perform discrete steps if possible.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

C99 code generation has the following additional restrictions:

- List comprehension expressions are not supported.

Differences in output

When converting a real number to text, for example when printing a real value, the generated C99 code does not preserve trailing zeroes of real number fractions while using `%g`. For example `fmt("%.3g", 1.0)` will output `1` rather than `1.000`.

This may cause real number values to be printed with less digits precision than indicated in the CIF program.

Strings have an upper limit in length that is set during compilation. Longer strings are silently truncated to this length.

Options

The C99 code generator only uses the common options that apply to all target languages/platforms of the CIF code generator, it has no options of its own.

The code prefix that can be configured using a common option is used to prefix external names, making them unique for the generated code.

Generated files

C99 code generation leads to creation of seven files:

Generated file	Purpose
`<prefix>_library.h`	Runtime support code and library headers
`<prefix>_library.c`	Runtime support code and library implementation
`<prefix>_engine.h`	Interface definition of the translated CIF program
`<prefix>_engine.c`	Implementation of the translated CIF program
`<prefix>_compile.sh`	Compile script for Linux
`<prefix>_test_code.c`	Example of external functions, allow compiling the generated code
`<prefix>_readme.txt`	Short description of use, the interface, and compile options

where `<prefix>` is replaced by the value of the *Code prefix* option during code generation.

The aim is to provide a complete and compilable package, to minimize the effort in converting a CIF specification. Most files are static text (modulo the `<prefix>` replacement). The two exceptions are both engine files that contain the C99 version of the CIF specification. The interface of these files is however fixed (modulo the `<prefix>` replacement), and generated data types and operations on them have predictable names, reducing the chance of failed compilation when re-generating the files.

When modifying a generated file (such as the compile script or the example test code), keep in mind that running the code generator again will overwrite *all* files. Rename a file before modifying it to avoid loss of work.

Compilation of the generated code

The generated code is C99 code that can be compiled using a C99 C compiler, like `gcc`:

```
gcc -Wall -std=c99 -DPRINT_OUTPUT=1 -DCHECK_RANGES=1 \
    <prefix>_engine.c <prefix>_library.c <prefix>_test_code.c -lm
```

The single `\` is not part of the command, it means the next line should be appended at the end.

The `<prefix>` should be replaced by the value of the *Code prefix* option during code generation. The `PRINT_OUTPUT` and `CHECK_RANGES` macros enable support for the CIF `print` statement and checking of integer range overlap, as defined by CIF. In general however, you may want to replace the test code file by your own code.

C99 compile-time options

Besides the `PRINT_OUTPUT` and `CHECK_RANGES` macros, the generated code has a few other compile-time options to customize its behavior. Below is the complete list.

Macro name	Effect
`MAX_STRING_SIZE`	CIF has variable length strings, in the generated code they are converted to strings with a fixed upper limit. Default is 128 (including the terminator character). Set this macro for a different limit. *Note that both the library and the engine use this macro. It is crucial that both files use the same value for this macro*.
`CHECK_RANGES`	When copying integers from one range to another, where the latter is only partially covered by the former, CIF checks whether the copied values do not violate the destination range. Since violating this property generally indicates a programming error, it is recommended to enable this macro.
`EVENT_OUTPUT`	If set, the generated code calls `<prefix>_InfoEvent()` before and after performing an event to allow third-party code to act on it. Enable if your code needs to know when an event is performed.
`PRINT_OUTPUT`	If set, the CIF `print` statement is performed, and the `<prefix>__PrintOutput()` is called with the resulting text and destination filename. Mainly intended for debugging, but only useful in systems where the text can be displayed to a user.
`MAX_NUM_EVENTS`	The CIF semantics state that execution of edges is instantaneous and time-less. The generated code loops to perform edges, until no edge can be executed any more. Unfortunately, such a loop is not time-less. To avoid requiring too much time, there is an upper limit of 1000 iterations. Set this macro to change the limit.
`KEEP_RUNNING`	The CIF language checks range conversions (if `CHECK_RANGES` is set) and verifies updates of continuous variable updates. Setting this flag disables a few checks. Most of these checks are in `assert()` checks and not controlled by this macro, since violating the condition generally means a crash is eventually inevitable. It is recommended to avoid using this macro.

Executing the code

The engine files act as a library that implements the CIF program. To run the program, the code must be called regularly. The code has two entry points, one for initializing and performing the first steps, and one for handling a time delay.

The first entry point is:

```
<prefix>_EngineFirstStep(void)
```

This entry point initializes all the data, queries the values of the input variables if present, and performs execution of edges until blocked on the first time step or until hitting the `MAX_NUM_EVENTS` limit. Since this resets the entire CIF program to its initial state, you should only call this when the system being controlled is also (re-)initialized.

The second entry point is more regularly called, after a period of time has passed:

```
<prefix>_EngineTimeStep(double delta)
```

By calling this entry point, you indicate that `delta` (> 0) time units have passed. The code reads new values of the input variables, updates the continuous variables, and performs edges until all edges are blocked, or hitting the `MAX_NUM_EVENTS` limit.

When the call of either entry point returns, you can query the value of the variables to use as input for other parts of the system.

Environment interface

Calling one of the above entry points causes the CIF program to be executed, up to the next time step (assuming the `MAX_NUM_EVENTS` limit is high enough). During the execution, the code performs a number of callbacks to get information from the environment, to provide information about its actions, or to deliver output. In addition, after each call, variables may be inspected by the environment to get information about the decisions of the CIF program.

The externally provided callback functions that are being used are:

```
<prefix>_AssignInputVariables(void)
```

The CIF program requests new values for all its input variables. In your implementation of this function you should write values directly in the input variables. Input variables are assumed to be independent of other parts of the CIF program, in general you should not need other CIF program values to implement this function.

If `EVENT_OUTPUT` is set during compilation, executed events are reported with:

```
<prefix>_InfoEvent(<prefix>_Event_ event, BoolType pre)
```

If the EVENT_OUTPUT macro is set during compilation, the generated code reports with this call that event is about to (pre is TRUE), or has been (pre is FALSE) executed. In the pre-event call, the event can be executed, and all variables have their value before performing the updates of the involved edges. In the post-event call, all variables have their values updated as indicated in the updates of the edges.

Primary use of this call is to forward the decision to other parts of the controlled system.

Finally, if the PRINT_OUTPUT macro is set during compilation, the generated code reports lines of output to be printed with the following function call:

```
<prefix>_PrintOutput(const char *line, const char *fname)
```

It denotes that text line should be printed at file fname. The primary uses of this call are to enable debugging the system, or to log relevant events. Note that fname here can mean anything. It can mean a real file, but also a pseudo-device like :stdout which is commonly used to print output to the C stdout stream.

Data access

Each CIF constant is available as a variable that does not change after initialization. Each algebraic variable can be queried by calling a function with the same name as the variable. The value of continuous variables is available by accessing the equivalent variable in the generated code. The derivative of each continuous variable is available as function named by the variable, and the suffix deriv. The model time is available as model_time. There is no derivative function of time. Internal functions are available as well if required.

The engine header file lists the translated names. It should be easy to find the relevant entry, as the generated code also documents the original name.

Data types

The following table defines how CIF data types are converted in the C99 generated code.

CIF data type	C99 data type
bool	BoolType which is a typedef to int
int	IntType which is a typedef to int
enum e = ...	<prefix>Enum which is a typedef to enum Enum<prefix>_
real	RealType which is a typedef to double
string	StringType which is a typedef to a struct
list[n] t	...Type which is a typedef to a struct
tuple(...)	...Type which is a typedef to a struct

A few notes:

- The `...` in the C99 `...Type` name for arrays (fixed length lists) and tuples is the systematic type name of the array or tuple type.
- Due to preprocessing all enumerations are merged into one. The generated C99 code always has exactly one translated `enum`.
- The elementary data types (`BoolType`, `IntType`, `RealType`, and `StringType`) are defined in the library header file.
- The `StringType` structure wraps a `char data[MAX_STRING_SIZE]` array. The `struct` allows copying arrays by value using an assignment statement.
- The array type (list with a fixed length) structure wraps a `<element-type> data[<size>]` array for the same reason. Copying an array by assignment is allowed.
- The tuple type is a structure with fields `_field0`, `_field1`, and so on, where the type of each field in the structure matches with the type of the CIF tuple field.
- Boolean constants `true` and `false` are named `TRUE` and `FALSE` respectively.
- The `-0.0` value of doubles gets replaced by `0.0` to void subtle equality problems.
- Strings get silently truncated to `MAX_STRING_SIZE - 1` characters. They are always terminated with a NUL character.
- Enumeration values get `_<prefix>_` added in front to make them unique.

Systematic type names

For tuple and list types, the type is converted to a name in a systematic way, making each type unique and predictable.

The conversion composes a name of a type from the type of its elements, and concatenates the result. The following conversion table applies:

CIF type	Systematic name
`bool`	`B`
`int`	`I`
`enum e = ...`	`E` (Due to linearization, all enumerations are merged together to a single enumeration. As such, always exactly one `enum` is generated for a CIF model.)
`real`	`R`
`list[n] t`	`A<n><t>`, with `<n>` the length of the array, and `<t>` the systematic type name of the element type of the array.
`tuple(t1, t2, t3, ..., tn)`	`T<n><t1><t2><t3>...<tn>`, with `<n>` the number of fields in the tuple, and `<t1>` to `<tn>` the systematic type names of the fields.

For example, type `list[5] int` is converted to `A5I`, and type `tuple(list[5] int x; tuple(real a, b) z)` is converted to `T2A5IT2RR` (concatenation of `T2` with the field type names `A5I` and `T2RR`).

Runtime errors

The CIF language defines strict sets of allowed values for all data types, and performs checking of these values at run time. The generated C99 code follows that idea, and checks whether operations on data are safe. Most of these checks are performed with an `assert`. Executed C code does not provide useful stack traces by default, and original line numbers of e.g. a CIF program have no meaning due to the linearization that is performed before code is generated. In addition, the generated code is quite readable and it is not too hard to understand what the code is doing at some C source line.

Java code generation

The CIF code generator can generate Java code from a CIF specification. It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to Java code generation.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

Java code generation has the following additional restrictions:

- List comprehension expressions are not supported.

Options

Besides the common options that apply to all target languages/platforms of the CIF code generator, the Java code generation can be influenced by the following additional options:

- *Java package*: The Java package in which to generate Java code. If not specified, the code prefix is used. The specified package should be a valid Java package name, and must not be empty.

The code prefix that can be configured using a common option, is used in the generated Java code as prefix for class names.

Compilation

The generated Java code can be compiled with a Java 7 compiler or newer.

The generated code is complete, and can be used as is, without any CIF related library or runtime. No additional dependencies or libraries are required. Only the standard Java libraries, part of any Java installation, are needed.

Generated files

Java code generation leads to two Java files being generated:

- `<prefix>.java`
- `<prefix>Test.java`

where `<prefix>` is replaced by the code prefix configured using the *Code prefix* option.

The first file contains the code for all the features of the CIF model in an abstract class named `<prefix>`. The second file contains a class named `<prefix>Test`, which inherits from the `<prefix>` class, and implements various abstract methods. This test class can be used to test the code, and can serve as a starting point for actually implementing the coupling between the code generated for the CIF model, and the environment in which it runs.

Executing the code

The code can be executed in multiple ways:

- `execOnce` method

  ```
  public void execOnce(double newTime)
  ```

  Invoke this method to execute the code once. Inputs are read, transitions are executed until none are possible, output is written, etc.

  The `newTime` is the time in seconds, since the start of the first execution of this method. For each invocation, the time that has elapsed so far should be provided. This becomes the new value of variable `time`.

  Using this method, you decide when you invoke this method, and with what values to use for variable `time`. This approach allows full control.

- `exec` method

  ```
  public void exec(long frequency)
  ```

  Invoke this method to execute the code repeatedly, without ever stopping. This repeatedly invokes the `execOnce` method.

  The `frequency` can be used to configure how often, in number of times per second, the code should be executed. By given a frequency of `100`, the method attempts to execute the code every 10 milliseconds. If a single execution takes less than 10 milliseconds, the method will sleep for the remainder of what is left of the 10 milliseconds. However, if a single execution takes more than 10 milliseconds, the next execution is immediately started.

  It is also possible to execute the code repeatedly, as fast as possible, without any delays. As soon as a single execution has finished, another execution is started. This kind of behavior can be obtained by providing a negative or zero value for `frequency`.

Environment interface

The code generated for the CIF model does not do anything useful by itself. Only once it is coupled to the environment, can the CIF model become useful. The following methods are provided in the interface of the generated class as abstract methods, and need to be implemented in a derived class:

- `updateInputs` method

```
protected void updateInputs()
```

In this method, you should update the values of the input variables of the CIF model. Note that you should obtain these from the environment. You should not access the state of the CIF model from this method, as it may not be initialized or up-to-date yet.

An example of how to update the input variables, can be found in the generated `<prefix>Test` class. There, each input variable is assigned the default value for its data type.

If you want to update the variable, you should always assign completely new fresh values. Don't modify arrays and tuples in-place, as that may lead to other variables being changed as well.

- `preExec` method

```
protected void preExec()
```

This method is invoked each time, just before the code for the CIF model is executed. This method is invoked before the `updateInputs` method is invoked.

You should not access the state of the CIF model from this method, as it may not be initialized or up-to-date yet.

- `postExec` method

```
protected void postExec()
```

This method is invoked each time, just after the code for the CIF model is executed. In this method, you should write the output variables of the CIF model to the environment. Since the code for the CIF model was just executed, these variables may have new values.

All discrete and continuous variables of the CIF model are available, as are variable `time` and the current locations of all of the automata with at least two location.

The code to write the output values to the environment is the dual to reading the inputs from the environment, as is done in the `updateInputs` method.

- `infoExec` method

```
protected void infoExec(long duration, long cycleTime)
```

This method informs about the duration of a single execution of the code generated for the CIF model. The `duration` is the total number of nanoseconds spent executing the code. The `cycleTime` is the desired maximum duration of the execution, in nanoseconds, or `-1` if not available.

You can use this method to detect when the code runs longer than the desired maximum duration, and thus the desired execution frequency can not be achieved. You can also use this

method to figure out the variability of the duration of execution.

The `doInfoExec` field of the class can be used to configure whether this method is invoked during execution by the `exec` method. This is enabled (`true`) by default.

- `infoEvent` method

  ```
  protected void infoEvent(int idx, boolean pre)
  ```

  This method informs about events that are about to be executed or have just been executed. The `idx` is the 0-based index of the event. You can feed this index to the `getEventName` method to obtain the absolute name of the event. The `pre` is `true` if the event is about to be executed, and `false` if the event has just been executed.

  You can use this method to be informed about what the code does during execution. You could for instance log that for debugging.

  The `doInfoEvent` field of the class can be used to configure whether this method is invoked during execution. This is disabled (`false`) by default.

- `infoPrintOutput` method

  ```
  protected void infoPrintOutput(String text, String target)
  ```

  This method informs that new print output has been generated during execution. The `text` is the text that is to be printed. The `target` indicates the file or special target to which text is to be printed. If printed to a file, an absolute or relative local file system path is given. Paths may contain both `/` and `\` as file separators. Supply the path to the `<prefix>Utils.normalizePrintTarget` method to normalize the path to use file separators for the current platform on which the Java code is executed.

  There are two special targets: `:stdout` to print to the standard output stream, and `:stderr` to print to the standard error stream.

  You should use this method to actually print the text to standard output, standard error, or files. An example of how to do this, can be found in the generated `<prefix>Test` class.

  The `doInfoPrintOutput` field of the class can be used to configure whether this method is invoked during execution. This is enabled (`true`) by default.

Data access and manipulation

For each CIF constant, a field is generated in the Java class. Similar fields are generated for the state variables (discrete and continuous variables), and the input variables. A `time` field is always present and contains the current model time.

For each internal user-defined function of the CIF model, a method is generated in the Java class. Similarly, a method is generated for each algebraic variable of the CIF model, as well as for the

derivative of each continuous variable (except variable `time`).

The standard library functions and operators that are supported by the code generation, are also available, in the `<prefix>Utils` inner class. Not all operators and standard library functions are available. Only those that have no corresponding Java operator or method, or behave differently with respect to runtime errors, are available in this inner class.

For more information on the names of the generated field and methods, see the section on naming below.

Data types

The following table lists CIF types supported by the code generator, and their equivalent in the generated Java code:

CIF type	Java type	Tuple postfix
`bool`	`boolean` / `java.lang.Boolean`	`B`
`int` (with or without range)	`int` / `java.lang.Integer`	`I`
`real`	`double` / `java.lang.Double`	`R`
`string`	`java.lang.String`	`S`
`list[...] t`	`java.util.List<t>`	`Lt`, with `t` a postfix for the element type
`enum e = ...`	`<prefix>Enum` inner enumeration (class)	`E`
`tuple(...)`	`CifTuple_T...` inner class	`TnFFF`, with `n` the number of fields, and `FFF` a postfix for the type of each field.

Notes:

- For each unique tuple type (ignoring integer and array ranges of the types of the fields, as well as field names) an inner Java class is generated. Their names all start with `CifTuple_`, and end in a postfix that describes the type. The *Tuple postfix* column in the table lists the texts used for each type in the postfix. For instance, for a `tuple(tuple(int x; real y; string z) a, list[3] int b)` type, the tuple class would be named `CifTuple_T2T3IRSLI`.
- For `boolean`, `int`, and `double`, the primitive type is preferred. It's boxed variant is used when needed, for instance for element types of a `java.util.List<...>` class.
- Generic lists are not supported, only CIF arrays can be used.
- Due to preprocessing all enumerations are merged together into a single enumeration. As such, always exactly one Java `enum` is generated for a CIF model.

Runtime errors

Some CIF models that are syntactically valid, may lead to runtime errors when simulated or executed. For instance, there may be a division by zero, an out of bounds projection, or an assignment may lead to the range of the assigned variable being violated. The generated Java code

detects all such problems and throws a `<prefix>Exception` in such cases. This exception class is an inner class of the `<prefix>` class. The exception indicates what caused the runtime error, using an informative end-user readable message.

Naming

The generated code will contain names for variables, functions, etc. The names in the generated code are based on the absolute names of the objects in the original CIF model. The names are further influenced by the linearization algorithm. Essentially, the absolute names are used, where each `.` is replaced by a `_`. Furthermore, a `_` is added at the end, to avoid conflicts with other non-generated names, Java keywords, etc. For instance, for an automaton `a` with a discrete variable `b` in it, the absolute name is `a.b`. In the Java code, the variable will be named `a_b_`.

The code generator ensures unique names in the generated Java code. If the same Java name results from two different CIF objects, one of them is renamed, by adding a `2`, or `3`, or `4`, etc to make it unique. For instance, if the CIF model has a constant named `a_b` in the top level scope, and a constant named `b` in the `a` automaton, both would be named `a_b_`. One of them is renamed to `a_b_2`. If such renaming takes place, a warning is printed to the console.

CIF model	Java code	Example CIF name	Example Java name
Constant	Field	`a.b`	`a_b_`
Discrete variable	Field	`a.b`	`a_b_`
Continuous variable	Field	`a.b`	`a_b_`
Derivative of a continuous variable	Method	`a.b'`	`a_b_deriv()`
Algebraic variable	Method	`a.b`	`a_b_()`
Input variable	Field	`a.b`	`a_b_`
User-defined function	Method	`a.b`	`a_b_()`
Parameter of a user-defined function	Method parameter	`a.b`	`a_b_`
Local variable of an internal user-defined function	Local variable of a method	`a.b`	`a_b_`
Enumeration	Enumeration	`a.b`	`<prefix>Enum`
Enumeration literal	Enumeration constant	`a.b.c`	`<prefix>Enum._c`
Tuple type	Inner class	n/a	`CifTuple_T...`

Notes:

- As linearization is applied on the CIF model prior to generating Java code, all enumerations are merged together to a single enumeration. As such, always exactly one Java `enum` is generated for a CIF model.
- For details on tuple types and the classes that are generated for them, see the data types section above.

JavaScript code generation



JavaScript code generation is currently an experimental feature.

The CIF code generator can generate JavaScript code from a CIF specification. It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to JavaScript code generation.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

JavaScript code generation has the following additional restrictions:

- List comprehension expressions for non-array lists are not supported.
- List comprehension expressions with a filter are not supported.

Options

The JavaScript code generator only uses the common options that apply to all target languages/platforms of the CIF code generator. It has no specific options of its own.

The code prefix that can be configured using a common option, is used in the generated JavaScript code as prefix for class names.

Compilation

The generated JavaScript code does not require compilation. The generated code is complete, and can be used as is, without any CIF related library or runtime. No additional dependencies or libraries are required. Only standard JavaScript functions, part of for instance any browser, are needed.

Generated files

JavaScript code generation leads to two files being generated:

- `<prefix>_class.js`
- `<prefix>_utils.js`

where `<prefix>` is replaced by the code prefix configured using the *Code prefix* option.

The first file contains the code for all the features of the specific CIF model, primarily in a class named `<prefix>_class`. The second file contains the code for utility methods, primarily in a class named `<prefix>Utils`. The functionality from the second file is used the first file.

Executing the code

The code can be executed in multiple ways:

- `execOnce` method

```
execOnce(newTime)
```

Invoke this method to execute the code once. Inputs are read, transitions are executed until none are possible, output is written, etc.

The `newTime` is the time in seconds, since the start of the first execution of this method. For each invocation, the time that has elapsed so far should be provided. This becomes the new value of variable `time`.

Using this method, you decide when you invoke this method, and with what values to use for variable `time`. This approach allows full control.

- `exec` method

```
exec()
```

Invoke this method to execute the code repeatedly, without ever stopping. This repeatedly invokes the `execOnce` method.

The `frequency` field (in the same class) can be used to configure how often, in number of times per second, the code should be executed. By given a frequency of `100`, the method attempts to execute the code every 10 milliseconds. If a single execution takes less than 10 milliseconds, the method will wait for the remainder of what is left of the 10 milliseconds. However, if a single execution takes more than 10 milliseconds, the next execution is immediately started.

It is also possible to execute the code repeatedly, as fast as possible, without any delays. As soon as a single execution has finished, another execution is started. This kind of behavior can be obtained by providing a negative or zero value for `frequency`.

The `speed` field (in the same class) can be used to configure the execution speed. With a speed of `1.0`, one unit of model time takes one second during execution. With a speed of `2.0`, one unit of model time takes half a second during execution. With a speed of `0.5`, one unit of model time takes two seconds during execution. And so on. The speed must be positive.

`execOnce` provides flexibility, but lacks automation for repeated execution. `exec` provides automatic repeated execution, but lacks flexibility. To support automatic repeated execution, with pausing/resuming support, and the ability to reset and restart the execution, use the following methods instead of `execOnce` and `exec`:

- `start` method

```
start()
```

Start or continue execution. If execution is already in progress, invoking this method has no effect.

- `stop` method

```
stop()
```

Stop or pause execution. If execution is not in progress, invoking this method has no effect.

- `reset` method

```
reset()
```

Invokes the `stop` method and resets the state. If execution is subsequently started via the `start` method, it starts from the initial state.

Environment interface

The code generated for the CIF model does not do anything useful by itself. Only once it is coupled to the environment, can the CIF model become useful. The following methods are provided in the interface of the generated class as methods that may be modified or provided a (different) implementation in a derived class:

- `updateInputs` method

```
updateInputs()
```

In this method, you should update the values of the input variables of the CIF model. Note that you should obtain these from the environment. You should not access the state of the CIF model from this method, as it may not be initialized or up-to-date yet.

If you want to update the variable, you should always assign completely new fresh values. Don't modify arrays and tuples in-place, as that may lead to other variables being changed as well.

- `preExec` method

```
preExec()
```

This method is invoked each time, just before the code for the CIF model is executed. This method is invoked before the `updateInputs` method is invoked.

You should not access the state of the CIF model from this method, as it may not be initialized or up-to-date yet.

- `postExec` method

```
postExec()
```

This method is invoked each time, just after the code for the CIF model is executed. In this method, you should write the output variables of the CIF model to the environment. Since the

code for the CIF model was just executed, these variables may have new values.

All discrete and continuous variables of the CIF model are available, as are variable `time` and the current locations of all of the automata with at least two location.

The code to write the output values to the environment is the dual to reading the inputs from the environment, as is done in the `updateInputs` method.

- `infoExec` method

```
infoExec(duration, cycleTime)
```

This method informs about the duration of a single execution of the code generated for the CIF model. The `duration` is the total number of milliseconds spent executing the code. The `cycleTime` is the desired maximum duration of the execution, in milliseconds, or `-1` if not available.

You can use this method to detect when the code runs longer than the desired maximum duration, and thus the desired execution frequency can not be achieved. You can also use this method to figure out the variability of the duration of execution.

The `doInfoExec` field of the class can be used to configure whether this method is invoked during execution by the `exec` method. This is enabled (`true`) by default.

By default, this method does not do anything.

- `infoEvent` method

```
infoEvent(idx, pre)
```

This method informs about events that are about to be executed or have just been executed. The `idx` is the 0-based index of the event. You can feed this index to the `getEventName` method to obtain the absolute name of the event. The `pre` is `true` if the event is about to be executed, and `false` if the event has just been executed.

You can use this method to be informed about what the code does during execution. You could for instance log that for debugging.

The `doInfoEvent` field of the class can be used to configure whether this method is invoked during execution. This is enabled (`true`) by default.

This method can print information about event transitions that are executed. The `doTransitionPrint` field of the class can be configured to enable or disable printing of transition output. By default, transition printing is enabled (`true`).

This method can print information about states that are reached. The `doStatePrint` field of the class can be configured to enable or disable printing of state output. The `doStatePrintAlgVars` field of the class can be configured to enable or disable printing of algebraic variables as part of the state, which only has an effect if printing state output is enabled. By default, state printing is

disabled (`doStatePrint` is `false`). By default, algebraic variables are excluded (`doStatePrintAlgVars` is `false`).

- `infoPrintOutput` method

  ```
  infoPrintOutput(text, target)
  ```

  This method informs that new print output has been generated during execution. The `text` is the text that is to be printed. The `target` indicates the file or special target to which text is to be printed. If printed to a file, an absolute or relative local file system path is given. Paths may contain both `/` and `\` as file separators. Supply the path to the `<prefix>Utils.normalizePrintTarget` method to normalize the path to use `/` file separators.

  There are two special targets: `:stdout` to print to the standard output stream, and `:stderr` to print to the standard error stream.

  The `doInfoPrintOutput` field of the class can be used to configure whether this method is invoked during execution. This is enabled (`true`) by default.

  By default, print output is printed to the JavaScript `console`. Output to files is also printed to the console, and is prefixed with the path.

Data access and manipulation

For each CIF constant, a field is generated in the JavaScript class. Similar fields are generated for the state variables (discrete and continuous variables), and the input variables. A `time` field is always present and contains the current model time.

For each internal user-defined function of the CIF model, a method is generated in the JavaScript class. Similarly, a method is generated for each algebraic variable of the CIF model, as well as for the derivative of each continuous variable (except variable `time`).

The standard library functions and operators that are supported by the code generation, are also available, in the `<prefix>Utils` class. Not all operators and standard library functions are available. Only those that have no corresponding JavaScript operator or method, or behave differently with respect to runtime errors, are available in this class.

For more information on the names of the generated field and methods, see the section on naming below.

Data types

The following table lists CIF types supported by the code generator, and their equivalent in the generated JavaScript code:

CIF type	JavaScript type	Tuple postfix
`bool`	`boolean`	`B`
`int` (with or without range)	`number`	`I`

CIF type	JavaScript type	Tuple postfix
`real`	`number`	`R`
`string`	`string`	`S`
`list[...] t`	`object` (an array)	`Lt`, with `t` a postfix for the element type
`enum e = ...`	`<prefix>Enum` inner class	`E`
`tuple(...)`	`CifTuple_T...` class	`TnFFF`, with `n` the number of fields, and `FFF` a postfix for the type of each field.

Notes:

- For each unique tuple type (ignoring integer and array ranges of the types of the fields, as well as field names) a JavaScript class is generated. Their names all start with `CifTuple_`, and end in a postfix that describes the type. The *Tuple postfix* column in the table lists the texts used for each type in the postfix. For instance, for a `tuple(tuple(int x; real y; string z) a, list[3] int b)` type, the tuple class would be named `CifTuple_T2T3IRSLI`.
- Due to preprocessing all enumerations are merged together into a single enumeration. As such, always exactly one JavaScript `enum` is generated for a CIF model.

Runtime errors

Some CIF models that are syntactically valid, may lead to runtime errors when simulated or executed. For instance, there may be a division by zero, an out of bounds projection, or an assignment may lead to the range of the assigned variable being violated. The generated JavaScript code detects all such problems and throws a `<prefix>Exception` in such cases. The exception indicates what caused the runtime error, using an informative end-user readable message.

Naming

The generated code will contain names for variables, functions, etc. The names in the generated code are based on the absolute names of the objects in the original CIF model. The names are further influenced by the linearization algorithm. Essentially, the absolute names are used, where each `.` is replaced by a `_`. Furthermore, a `_` is added at the end, to avoid conflicts with other non-generated names, JavaScript keywords, etc. For instance, for an automaton `a` with a discrete variable `b` in it, the absolute name is `a.b`. In the JavaScript code, the variable will be named `a_b_`.

The code generator ensures unique names in the generated JavaScript code. If the same JavaScript name results from two different CIF objects, one of them is renamed, by adding a `2`, or `3`, or `4`, etc to make it unique. For instance, if the CIF model has a constant named `a_b` in the top level scope, and a constant named `b` in the `a` automaton, both would be named `a_b_`. One of them is renamed to `a_b_2`. If such renaming takes place, a warning is printed to the console.

CIF model	JavaScript code	Example CIF name	Example JavaScript name
Constant	Field	`a.b`	`a_b_`
Discrete variable	Field	`a.b`	`a_b_`

CIF model	JavaScript code	Example CIF name	Example JavaScript name
Continuous variable	Field	`a.b`	`a_b_`
Derivative of a continuous variable	Method	`a.b'`	`a_b_deriv()`
Algebraic variable	Method	`a.b`	`a_b_()`
Input variable	Field	`a.b`	`a_b_`
User-defined function	Method	`a.b`	`a_b_()`
Parameter of a user-defined function	Method parameter	`a.b`	`a_b_`
Local variable of an internal user-defined function	Local variable of a method	`a.b`	`a_b_`
Enumeration	Enumeration class	`a.b`	`<prefix>Enum`
Enumeration literal	Object key	`a.b.c`	`<prefix>Enum._c`
Tuple type	Tuple class	n/a	`CifTuple_T...`

Notes:

- As linearization is applied on the CIF model prior to generating JavaScript code, all enumerations are merged together to a single enumeration. As such, always exactly one JavaScript `enum` is generated for a CIF model.
- For details on tuple types and the classes that are generated for them, see the data types section above.

HTML code generation



HTML code generation is currently an experimental feature.

The CIF code generator can generate HTML from a CIF specification. For a CIF specification, a single HTML file is generated, which can be opened in a browser. In this way, the CIF specification can be executed in a browser, similar to using the CIF simulator to simulate the specification (especially when enabling the simulator's execution mode).

The HTML file that is generated contains JavaScript code that is used to execute the CIF specification (see JavaScript code generation for more information). The HTML file also includes a simple UI, to control the execution. Furthermore, the SVG images coupled to the specification are included. The images are animated based on the state of the CIF model, if the specification contains CIF/SVG output mappings. The images can be clicked to interact with the specification and influence the execution, if the specification contains CIF/SVG input mappings.

It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to HTML code generation.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

HTML code generation has the following additional restrictions:

- List comprehension expressions for non-array lists are not supported.
- List comprehension expressions with a filter are not supported.

HTML page

Generated HTML files can be opened in any modern browser. The HTML files may not function correctly when opened in the browser embedded in the Eclipse ESCET IDE. They should therefore be opened in an external browser.

Areas

The HTML page primarily consists of two areas:

- In the upper area, the name of the model and the SVG images are displayed.
- In the lower area, several buttons and a log panel are present.

The lower area can be resized. If the log panel is not hidden, a small triangular marker appears at the bottom-right corner of the upper area, just above the *Hide* button. Press and drag the marker up or down to resize the lower area.

Execution

When the HTML file is opened in a browser, the execution start directly. Several buttons are present to control the execution:

- To stop the execution, press the *Stop* button.
- To continue the execution after stopping it, press the *Start* button.
- The execution can be reset to the start state by pressing the *Reset* button. Resetting the execution also stops it.

Visualization

If the specification features SVG images, these are displayed in the upper area. As the execution progresses, the images are updated based on the state of the CIF specification being executed. The visualization can be influenced using settings.

The order in which the SVG images are displayed is based on the sorted file names of the SVG image files, as included in the CIF specification. If some SVG image files have the same file names (in different directories), they are instead ordered based on their absolute paths.

Interaction

If the specification features CIF/SVG input mappings, the SVG images become interactive. Clicking an interactive element of an SVG image influences the execution of the specification.

Elements of the SVG images that are interactive get a red outline when hovering over them. The mouse pointer then also changes, to indicate that they can be clicked. When such an element is

clicked, the red outline changes to a green outline. The outlines can be configured using the *SVG input outline style* and *SVG input outline filters* options.

Clicking an interactive SVG element has no effect while the execution is stopped.

Log panel

In the lower area also a log panel is displayed. It provides a log of the current execution, such as which transitions are executed. Upon resetting the execution, the log is cleared.

The log panel can be hidden by pressing the *Hide* button. Once it is hidden, it can be shown again by clicking the *Show* button.

The log can be downloaded as a text file by pressing the *Download log* button. The filename of the downloaded log includes the model name, and a timestamp, such that different logs can be easily distinguished.

Settings

The settings menu can be shown by pressing the *Settings* button. Pressing the button again hides the settings menu. The following settings can be configured in the settings menu:

- **Frequency**

  A slider is present to set the frequency at which the code is executed. For instance, a frequency of 60 Hz means that the JavaScript code generated for the CIF specification is executed 60 times per second.

  Decreasing the frequency reduces the CPU usage of the browser, and makes animations less fluent. Conversely, increasing the frequency makes animations more fluent, but increases CPU usage.

  For more information, see the separate section on frequencies.

- **Speed**

  A slider is present to set the speed at which model time progresses. The configured speed is the amount of model time that elapses during a real-world second. For instance, with a speed of *1.0*, one unit of model time takes one second during execution, running at normal speed. With a speed of *2.0*, one unit of model time takes half a second during execution, making the execution twice as fast. With a speed of *0.5*, one unit of model time takes two seconds during execution, making execution half as fast.

  For more information, see the separate section on speeds.

- **Show image**

  In case the CIF specification is connected to multiple SVG images, a checkbox is present for each image.

  Checking a checkbox ensures that the corresponding image is shown. Unchecking a checkbox

hides the corresponding image.

By default all images are shown.

- **Log states**

  Whether to log all states that are reached. Is disabled by default.

  - **Include algebraic variables**

    Whether to include algebraic variables when logging states. Is disabled by default. Only has an effect if *Log states* is enabled.

- **Log transitions**

  Whether to log event transitions that are executed. Is enabled by default.

Options

The code generator has several options that are specific to HTML code generation. See the following sections for more information about these options and how to use them:

- HTML frequencies
- HTML speeds
- Choosing a frequency and speed
- SVG input outline style
- SVG input outline filters

HTML frequencies

The *HTML frequencies* option can be used to configure the minimum, default, and maximum frequencies in hertz (or Hz), the number of times per second that the JavaScript code for the model is executed in generated HTML pages. First the minimum frequency is given (must be at least 1), then a colon (`:`), then the default frequency (must be at least the minimum frequency and at most the maximum frequency), then another colon (`:`), and then the maximum frequency (must be greater than the minimum frequency, and at most 100,000). By default, the minimum frequency is 1 Hz, the default frequency is 60 Hz and the maximum frequency is 120 Hz.

In generated HTML pages, the current frequency can be set to different values (within the configured range), using a slider. Next to the slider, the current frequency is displayed. Changing the frequency can have the following consequences:

Decreasing the frequency	**Increasing the frequency**
Lower browser CPU usage	Higher browser CPU usage
Less fluent animations	More fluent animations
Less accurate time-based calculations	More accurate time-based calculations

HTML speeds

The *HTML speeds* option can be used to configure the range of speeds at which the code generated for the model can be executed in generated HTML pages. A speed is the amount of model time that elapses during a real-world second. Specify the minimum, default, and maximum speeds as numbers, separated by colons; for example `0.3:1:2`. The minimum speed must be at least 0.001 and at most 1. The default speed must be at least the minimum speed and at most the maximum speed. The maximum speed must be greater than the minimum speed, at least 1, and at most 1000. By default, the minimum speed is 0.1, the default speed is 1 and the maximum speed is 10.

In generated HTML pages, the current speed can be set to different values (within the configured range), using a slider. Next to the slider, the current speed is displayed. Changing the speed can have the following consequences:

Decreasing the speed	**Increasing the speed**
More fluent animations	Less fluent animations
More accurate time-based calculations	Less accurate time-based calculations

Choosing a frequency and speed

For a specification with a continuous variable (such as a timer), if calculations need to be more precise, such as detecting when a timed guard becomes enabled, increase the frequency and/or decrease the speed.

In general, if you want a precision in terms of `dt` time units, and the derivative of a continuous variable is `dy`, then you need `dy` / `dt` as minimum frequency, at a speed of 1.0. For instance, if `dt` is 0.1 and `dy` is 2, you need 2 / 0.1 = 20 as minimum frequency. However, the execution scheme may influence this, requiring you to use a higher frequency.

If you use a speed that is not 1.0, then you generally need to multiply the frequency by the speed. That is, if the simulation is twice as fast, you may need twice the frequency to preserve the precision of the simulation. Conversely, if the simulation is twice as slow, you could consider using half the frequency, while still preserving the precision of the simulation.

SVG input outline style

The style for the outline of interactive elements can be changed using the *SVG input outline style* option. Specify `off` to not show any outline. Specify a hover color, click color and stroke width, separated by colons, to specify a specific style. For colors, there are two ways to specify them. The first way to specify a color is to use one of the named colors: `black`, `blue`, `cyan`, `dark-gray`, `gray`, `green`, `light-gray`, `magenta`, `orange`, `pink`, `red`, `white`, or `yellow`. The second way to specify a color is to indicate an RGB color, by specifying comma-separated red, green and blue components, each in range [0..255]. For the stroke width, specify the width in pixels as a positive floating point number. Some examples of valid option values are: `blue:magenta:2.0`, `255,0,128:black:0.5`. The default value is `red:green:1.5`.

Disabling the outline does exactly only that: it prevents the outlines from being shown. It does not prevent the mouse pointer from being changed. Also, interactive elements can still can be clicked and this will still have the same effect.

SVG input outline filters

The outlines of interactive elements, if not disabled using the *SVG input outline style* option (using its `off` value), can be selectively disabled for certain interactive SVG elements, using the *SVG input outline filters* option. By default, all interactive SVG elements get an outline when hovering or clicking them. By specifying filters, you can change the default. For this option, specify comma-separated node names of SVG elements for which to show the outline. Note that you must specify node names of the elements, as specified in the SVG standard. So, not the ids of the elements that you can specify yourself. For example, use option value `circle,rect,text` to only show outlines for circles, rectangles and texts. You can use `*` in a filter to match zero or more characters of an element name. Prefix a filter with `-` to exclude elements with matching node names, rather than including them. Initially, before applying any filters, all elements get an outline, as if `*,` if prefixed to the filters. The filters are applied in the order they are specified. As an example, specify `*,-text` or simply `-text`, to give all interactive elements an outline, except for SVG `text` elements.

Disabling the outline does exactly only that: it prevents the outlines from being shown. It does not prevent the mouse pointer from being changed. Also, interactive elements can still can be clicked and this will still have the same effect.

Simulink C S-Function code generation

Everywhere at this page, the terms *Matlab*, *Simulink*, *S-Function*, and *Mex* refer to products and programs developed and owned by *MathWorks*. For more information, please visit https://mathworks.com/.

The CIF code generator can generate Matlab Simulink S-Function C code from a CIF specification. It is assumed the reader of this page is familiar with the general information of the CIF code generator tool. This page describes specific information applicable only to Simulink code generation.

The aim of the conversion from CIF to Simulink is to provide a smooth path from a CIF specification to a controlled real-time system.

At this page, all the details about the Simulink code generation are discussed. For an example of how to use this code generation in practice, read the example.

Supported specifications

The CIF code generator supports a subset of CIF specifications. See the Supported specifications section of the CIF code generator page for more information.

Simulink S-Function C code generation has the following additional restrictions:

- The type of CIF input variables is restricted, due to limitations in Simulink input ports.

  Simulink input ports can only handle real and complex signals. The latter cannot be expressed in CIF, while the former creates the need for CIF input variables with data types that can be relatively easily converted to real number values. For this reason, CIF input variables must have a boolean, enumeration, integer, or real elementary type. Arrays (fixed length CIF lists) of these

elementary types are also supported, they are converted to a vector of real numbers at a Simulink input port. Finally, arrays of arrays of the elementary types are converted to a matrix of real numbers at a Simulink input port.

Data types of all other CIF variables have no additional constraints. However, Simulink poses similar restrictions to its other data vectors, in particular the output ports. The CIF to Simulink code generator tool only makes variables available for output that are compatible with real number, vectors of real numbers, or matrices of real numbers. That means that use of the CIF string type, the tuple type, or arrays with a deeper nesting than two levels will not be available at the output.

- List comprehension expressions are not supported.

Differences in output

When converting a real number to text, for example when printing a real value, the generated Simulink code does not preserve trailing zeroes of real number fractions while using `%g`. For example `fmt("%.3g", 1.0)` will output `1` rather than `1.000`. This may cause real number values to be printed with less digits precision than indicated in the CIF program.

Strings have an upper limit in length that is set during compilation. Longer strings are silently truncated to this length.

The CIF print statement is supported, but only for the `:stdout` file. It is translated to the `ssPrintf` C function. This is also the only way to get a string to appear outside the Simulink S-Function block. Note that Simulink may drop the output function for platforms that have no output possibilities.

Options

The CIF to Simulink code generator uses the common options that apply to all target languages/platforms of the CIF code generator. It also has the additional options:

- *Simulink Outputs*: This option takes a list of CIF variables that should be made available as output of the generated S-Function. Only location variables (automata), discrete variables, continuous variables, and algebraic variables can be selected. The option also allows to use the wildcard character `*`, which represents zero or more characters from the name. You can use it to match several variables at once, for example `*out` matches all variables that end with `out`. The special name `nothing` means that no output should be produced at all.
- *Simulink sample time*: This option controls how Simulink samples the generated block. Allowed values are `continuous`, `inherited`, `variable`, or a sample period. Check the MathWorks `ssSetSampleTime` function documentation for details which to use. By default, sample time is set to `continuous`.
- *Simulink sample offset*: This option controls the offset of sampling. Its allowed values depend on the value of the *Simulink sample time* option. Possible values are `fixed`, `0.0`, or a positive non-zero value. Check the MathWorks `ssSetSampleOffset` function documentation for details which to use. By default, this options is set to `0.0`.

The code prefix that can be configured using a common option is used as name for the generated S-Function.

Generated files

Running the Simulink code generator produces two output files:

Generated file	Contents
`<prefix>.c`	The generated Simulink S-Function C code.
Report file	Links CIF variables to Simulink entries.

where `<prefix>` is replaced by the value of the *Code prefix* option during code generation.

The `<prefix>.c` file contains the source code of the actual S-Function. The file can be compiled using the Matlab Mex C compiler. The resulting mex file can be used in Simulink (Open the **Simulink Library**, select **User-defined functions**, and then the **S-Function** block, drag it into the Simulink canvas, open the block, and type the `<prefix>` name into the *Name* field.)

The report file lists how CIF variables got mapped onto the input, mode, contstate, and output Simulink vectors. You can use this information to connect your CIF program with the other parts of the Simulink model.

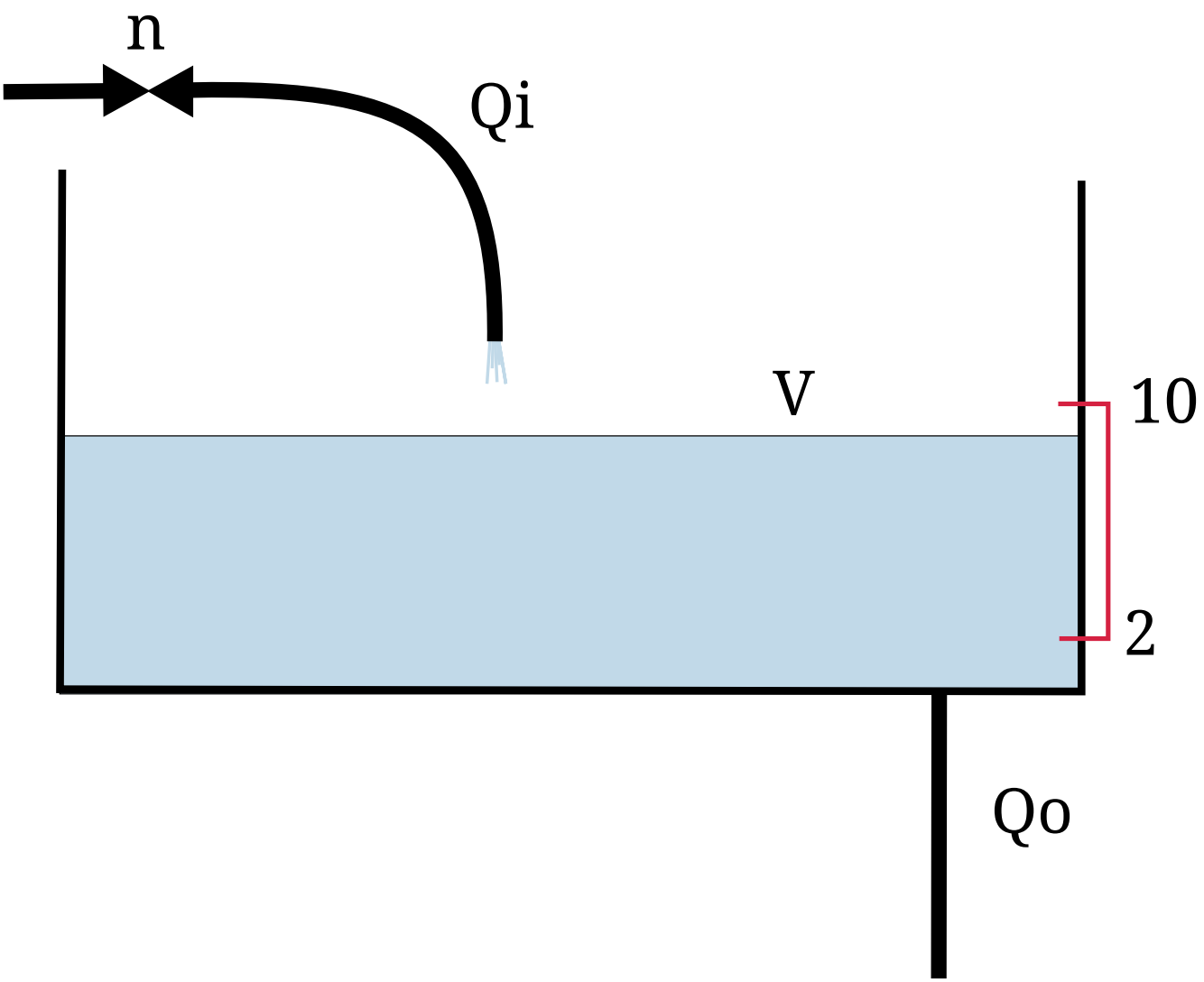
Simulink code generation and simulation example

The Simulink C S-Function code generation can generate Simulink C code from a CIF specification. Here an example is given to demonstrate how the code generator can be used to control a model in Simulink.

Matlab and *Simulink* are products owned by MathWorks.

System to control

The system to control is a simple water tank, as shown below.



The water in the tank has volume `V`. At the bottom of the tank is a pipe where water flows away with speed `Qo`. At the top is a pipe with valve `n` that can be open or closed. The `Qi` variable denotes the speed of water flowing into the tank. The equations that hold in this system are:

```
Qi = n * 5
Qo = sqrt(V)
V' = Qi - Qo
```

By switching `n` on or off, a controller can control the amount of water flowing into the tank, and with that, control the volume `V`. The objective is to keep the volume between 2 and 10.

Controller in CIF

A switching controller will be sufficient to control this system. In CIF that results in a `tank_ctrl.cif` file containing:

```
event open_valve;

automaton controller:
    disc int n = 0;
    input real V;
```

```
    location closed:
        initial;
        edge open_valve when V <= 2 do n := 1 goto opened;

    location opened:
        edge when V >= 10 do n := 0 goto closed;
end
```

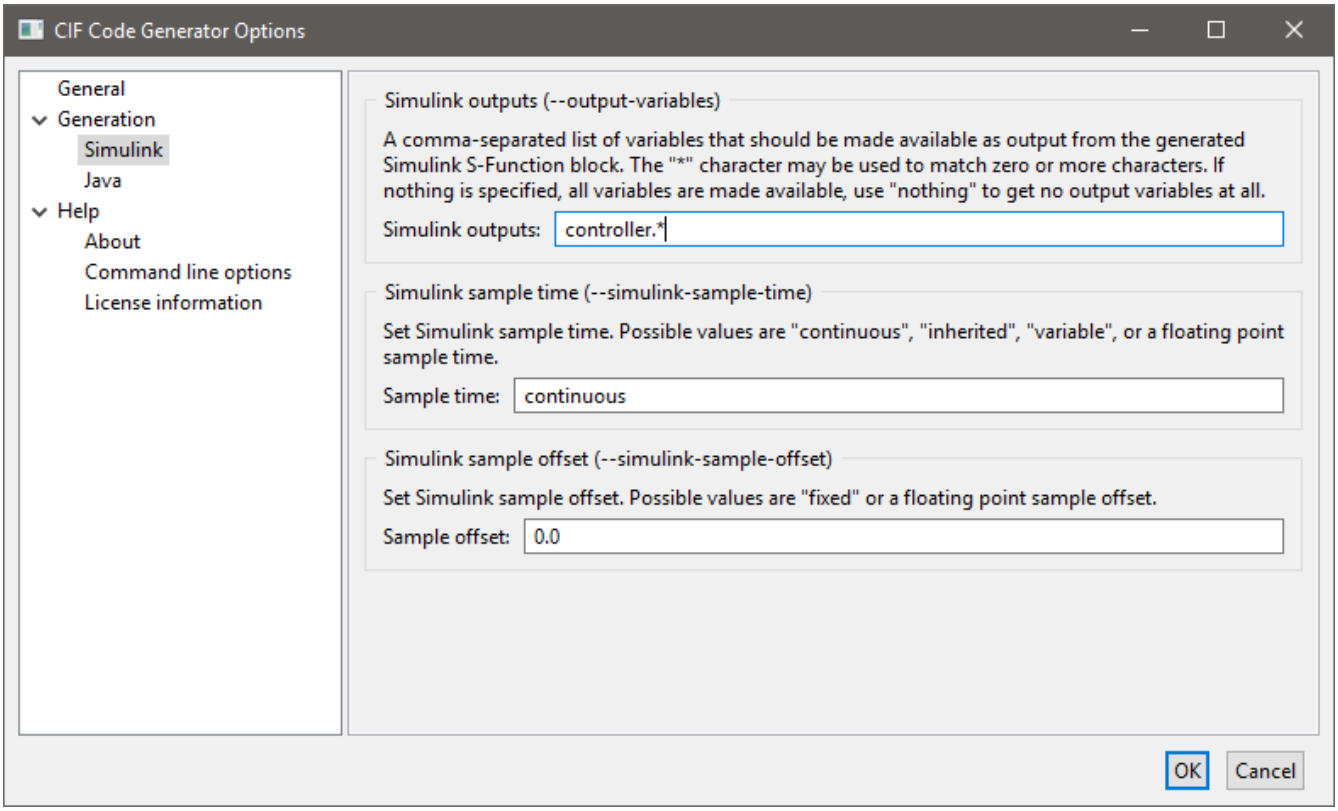
The controller has the control variable `n`, and input variable `V`. When the volume of the tank is below 2, `n` is set to `1`, which turns on the inflow. When the volume of the tank is high enough, `n` is set to `0` which turns the inflow off.

Generating Simulink C code

To simulate the controlled system in Simulink, the code generator tool can produce C code for Simulink that can be used in a Simulink *System SFunction* block. Convert the CIF model to C for Simulink with the code generation tool, by selecting the code generation tool from the menu under the right mouse button:



By default, you will get all internal variables at the output of the SFunction, which is often too much. The *Simulink Generation* tab allows precise definition of what output variables should be added, in this case all variables in the `controller` automaton. (Input variables are never made available from the CIF SFunction block, as they are already available in Simulink.)



After clicking **[OK]**, the CIF model is translated to C code. Code generation from the `tank_ctrl.cif` files results in two output files. The `tank_ctrl.c` file contains the generated C source code. The `tank_ctrl_report.txt` is a text file describing the data of the generated SFunction code:

```
Input/output report of the tank_ctrl.cif SFunction.

During code generation, CIF variables are made available in the Simulink vectors.
This report lists the variables in each vector, along with their index number.

Modes
-----
controller 1

Continuous states
-----------------
time 1

Inputs
------
controller.V 1

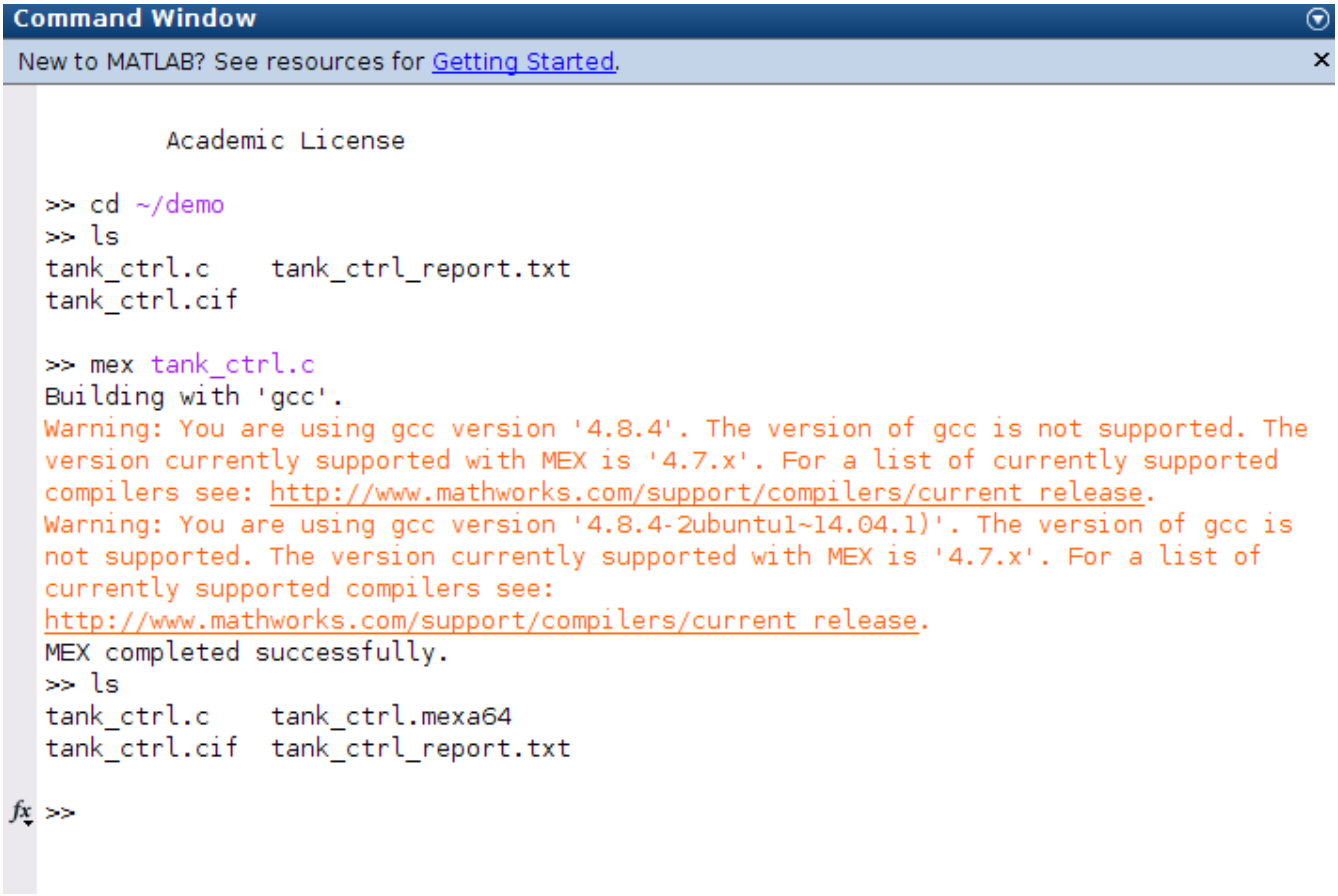
Outputs
-------
controller.n 1
```

The report shows which CIF variables are available at each vector. The *Inputs* and *Outputs* are the most interesting. Here it shows that `V` is the only input variable, `n` is the only output variable, and they are both the first value in their vector. This is what you would expect for such a small system. With bigger systems, the index numbers are much more useful for making proper connections in Simulink.

And this concludes the CIF part of this example, next step is setting things up in Matlab/Simulink.

Simulation in Simulink

After starting *Matlab*, and moving to the correct directory, you get a display like:



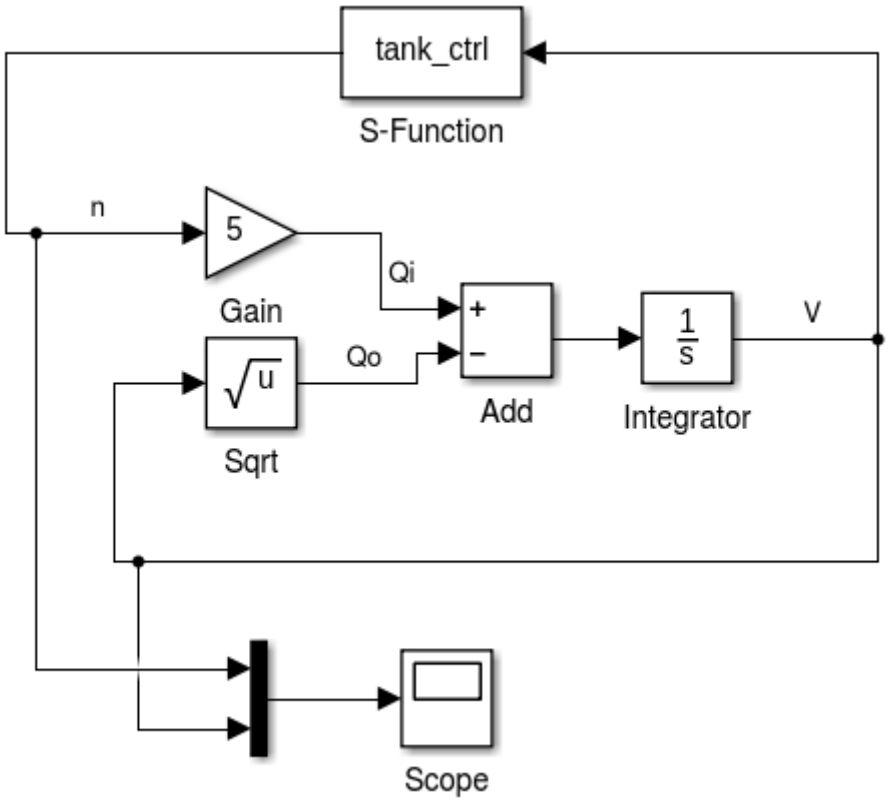
You can see the files present in the directory, as you would expect. In Matlab, the generated C code is compiled with the Matlab *Mex* compiler:

```
mex tank_ctrl.c
```

Here, the compiler prints a warning about the wrong compiler version, this may be different at your system.

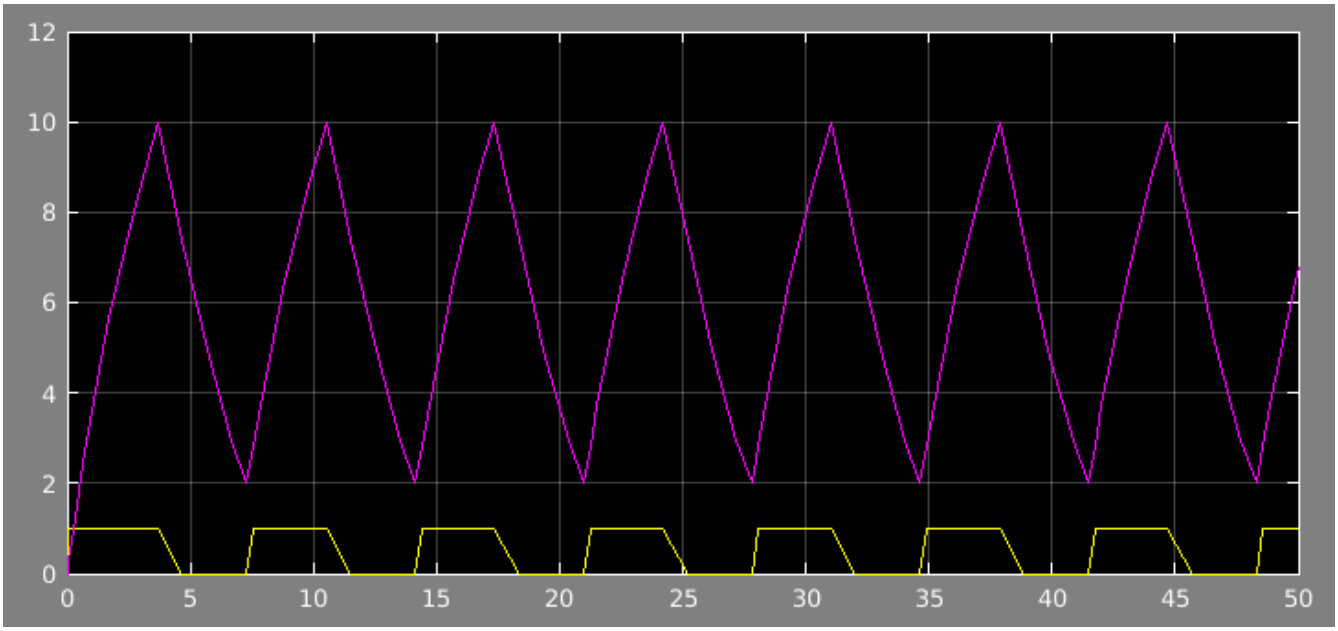
The result of the compilation is a `tank_ctrl.mexa64` file at a 64 bit system. If you use a 32 bit system, the filename extension is different.

The compiled CIF controller can be used in a Simulink *System SFunction* block, as shown below.



In the middle the equations are modeled. The output of the equations is `V`, which is fed into the `S-Function` block with the CIF controller at the top. The controller computes `n` which is an input for the equations.

At the bottom, a scope is attached that produces the following picture.



The volume fluctuates between 2 and 10, at the bottom is the control signal `n`.

4.5.2. CIF PLC code generator (stable)

The CIF PLC code generator takes a CIF specification and creates real-time **P**rogrammable **L**ogic **C**ontroller (PLC) code that can read values from external inputs, evaluate the state and perform event transitions, and finally write values to external outputs. The resulting code is written in the Structured Text programming language, one of the programming languages that is part of the IEC 61131-3 standard. The code adheres to the execution scheme defined by the CIF controller properties checker.

The generated code is not intended to be modified directly. Instead, the original CIF models should be modified, and from them new PLC code should be generated.

To successfully deploy the generated code at a real system, the CIF model should satisfy certain conditions and the used PLC should have a sufficiently short cycle-time to catch all input changes.

Starting the generator

The CIF PLC generator can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF implementation tools › Generate PLC code for CIF specification... (stable)**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF implementation tools › Generate PLC code for CIF specification... (stable)**.
- Use the `cifplcgen` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifplcgen` command line tool.

Options

Besides the general application options, this application has the following options:

Basic settings of the CIF PLC code generator

As different PLC systems have different expectations and limitations in what they accept as input, behavior of the PLC code generator heavily depends on the chosen PLC target. This is configured with the *PLC target type* option:

- *PLC target type*: The PLC code target type specifies the form of the desired output for the PLC model that eventually will run the generated code. Default value is *PLCopen XML*.

Other important settings are the *Input file* path for the CIF specification, the *I/O table file* path to the file that contains the connections to the external inputs and outputs, and the *Output file* for writing the generated PLC code:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *I/O table path*: The absolute or relative file system path to the I/O table path file that contains the I/O connections. For more information, consult the I/O table section. Default path is derived

from the CIF *Input file* option value. The `.cif` file extension is removed if it exists. Then, a `.csv` file extension is appended.

If the file path does not exist, code generation will continue with a warning that the generated code will not perform I/O with its environment.

- *Program header text file path*: The absolute or relative local file system path to the text file to include at the start of the generated PLC program. The program recognizes text patterns `${app-name}`, `${app-version}`, `${time-stamp}` and `${brief-explanation}`. They get replaced by the name of the PLC code generator application, its current version number, the current time/date and a short explanation of how to interpret CIF terminology in the PLC program, respectively. Non-printable and non-ASCII characters are ignored, `(*` is replaced by `(-*`, and `*)` is replaced by `*-)`. If no file path is provided, the program uses the following as input for the program header text:

  ```
  This file is generated with CIF's PLC code generator from the Eclipse ESCET
  toolkit.

  Generator name:    ${app-name}
  Generator version: ${app-version}
  Generation time:   ${time-stamp}

  ${brief-explanation}
  ```

- *Output file*: The absolute or relative local file system path to the output file or directory. If not specified, the output path will depend on the *Input file* option value. The result also depends on the chosen target type.

Tailoring the output of the code generator

Additional options exist to tailor the output of the code generator:

- *Event transitions form*: Select the desired form of the generated event transitions. Code generated by the PLC code generator executes the CIF specification by performing event transitions, similar to how the CIF simulator does (especially when enabling the simulator's execution mode). That part of the code can be generated in two different forms:

  - Use *All event code in main program* to obtain all event code directly in the main program,
  - Use *Event code is combined into a function by event controllability and CIF scope* to obtain event functions for all (un)controllable events in each scope, or
  - Use *Each event in a separate function* to obtain an event function for each event.

  The main program calls the event functions as needed. The former avoids function calls while the latter splits the code in several smaller functions. Which form is better depends on the target as well as the program.

- *Input/output code form*: Select the desired form of the generated input/output code. The generated code to read values from external input and the generated code to write values to external output can be created in the following forms:

    - Use *Both input and output code are in the main program* to obtain the code that reads input and the code that writes output directly in the main program,
    - Use *Input code in the main program and output code in a function* to obtain the code that reads input in the main program and the code that writes output in a function,
    - Use *Input code in a function and output code in the main program* to obtain the code that reads input in a function and the code that writes output in the main program, or
    - Use *Both input and output code each in their own function* to obtain the code that reads input in a function and the code that writes output in a function.

  The main program calls the input and/or output function as needed. Moving input and/or output code to function(s) reduces the size of the main program at the cost of one or two function calls. Which form is better depends on the target as well as the program.

- *Simplify values*: Enable this option to simplify values in the generated PLC code. The resulting code expresses more directly what values it actually uses as the cost of less abstractions in the generated code. This may make reviewing more difficult. This option is by default enabled.
- *Rename warnings*: Names in the CIF specification are re-used to improve linking generated PLC code back to the CIF file. However, as PLC code has different requirements for names, the names from the CIF specification are not always used as-is. By enabling this option, the generator prints a warning whenever a CIF name is not used as-is in the PLC code. This allows to obtain the original CIF name for the adapted name that is used in the generated PLC code. For more information, see the section about names. This option is by default disabled.

Overriding target type choices

Selecting the target type is usually enough to get PLC code for that PLC model. If that is not sufficient, the PLC code generator also allows overriding choices to tailor the generated PLC code.

Note that these options do not take limitations of the PLC model into account. An incorrect choice in these options may thus make the generated PLC code unusable for the specified target type. When using these options, always make sure the intended PLC model supports the configured values.

- *Convert enumerations*: Enumerations may have been used in the CIF specification. In addition, the PLC code generator introduces enumerations in its conversion of automata. This option controls how all enumerations are converted to PLC code:
    - Use *Automatically decide how to handle enumerations* (`--convert-enums=auto`) to let the code generator decide based on the configured target PLC model. This is the default value of the option. It will either preserve enumerations if the PLC model supports them, or it will convert the enumerations by one of the alternatives described below.
    - Use *Preserve enumerations* (`--convert-enums=keep`) if the target PLC supports enumerations. It is generally the preferred value because using enumeration names improves readability of the generated PLC code.
    - Use *Convert enumerations to constants* (`--convert-enums=consts`) if the target PLC does not support enumerations but does support constants. This option value also causes generation of names in the PLC code, but information about which names belong together is lost.
    - Use *Convert enumerations to integers* (`--convert-enums=ints`) if the target PLC neither

supports enumerations nor supports constants. This option value causes numeric values to be generated for values of the enumerations. That may cause difficulties in reviewing the generated PLC code.

- *PLC maximum iterations*: This option decides how maximum iteration loop limits are applied in the generated PLC code. In a CIF specification, performing events has priority over delaying in time. To get the same behavior as the CIF specification in the generated PLC code, the generated PLC code must also give priority to performing events. As a PLC is not fast enough to indefinitely perform events before the end of a PLC cycle, it is paramount that the PLC code will always eventually run out of events to perform. The bounded response property should hold for the specification to ensure that this is always the case. This property can be checked using the controller properties checker.

  The option takes two loop limits. The first loop limit defines the limit for uncontrollable events, while the second loop limit defines the limit for controllable events. There are three kinds of loop limits:

  - The limit `ctrl-props-anno` means that the PLC code generator derives the loop bound limit from the controller properties annotation in the specification. That is, the bound of the bounded response property is used, incremented by one to check at runtime that the bound is indeed sufficient. This is recommended, as it is the lowest loop limit that preserves CIF behavior in the generated PLC code. If the annotation is not available, or the specification does not have bounded response, the `inf` limit is used instead.
  - The limit `inf` means 'infinite'. The PLC code keeps trying to perform events, until no more (uncontrollable or controllable) events are enabled.
  - A limit consisting of a numeric value expresses the maximum number of loop iterations for deciding that all possible (uncontrollable or controllable) events have been done. It should be a positive number of at least `1`. Also, it should be large enough such that the PLC program will always run out of events it can perform, before reaching the loop limit. That is the case when the last iteration does not perform an event. If a lower value is used, the amount of processing time used by the PLC system in a single cycle to compute a response can be reduced. That results in a more timely response of the PLC, but it may not always compute the entire response. Whether this is acceptable should be decided on a case-by-case basis. Section Resulting PLC code explains how this affects processing.

  For example, if `20,inf` is given to the option, the PLC system will allow at most 20 loop iterations for uncontrollable events and it does not enforce a limit for controllable events. The default value of this option is `ctrl-props-anno,ctrl-props-anno`.

- *PLC task name*: The name of the generated PLC task that runs the converted CIF specification. If not specified, the option defaults to `PlcTask`.
- *PLC project name*: The name of the generated PLC project. If not specified, defaults to `Untitled1`.
- *PLC configuration name*: The name of the generated PLC configuration. If not specified, the option defaults to `Untitled1`.
- *PLC resource name*: The name of the generated PLC resource. If not specified, the option defaults to `Untitled1`.
- *PLC task cycle time*: If periodic task scheduling is desired, set the scheduling interval in

milliseconds for the task in the generated PLC code. To disable periodic task scheduling, disable the option. This option takes positive integer values and defaults to periodic task scheduling with a cycle time of 10 milliseconds.

- *PLC task priority*: The scheduling priority of the task to generate. The priority must be in the range [0 .. 65535], where priority 0 is the highest priority, and priority 65535 is the lowest priority. The priority can be used for preemptive or non-preemptive scheduling. If not specified, the priority defaults to 20.
- *PLC integer type size*: The number of bits of an integer value in the PLC. The default value of this option lets the target type decide. In general that is 32 bit, which is also the size of an `int` in CIF. If the PLC supports only smaller integer values, the largest available size is chosen.
- *PLC real type size*: The number of bits of a floating point value in the PLC. The default value of this option lets the target type decide. In general that is 64 bit, which is also the size of a `real` in CIF. If the PLC supports only smaller floating point values, the largest available size is chosen.

Supported specifications

The CIF PLC code generator assumes that the given CIF file has the following properties:

- It does not suffer from runtime evaluation errors. For instance, it should not be prone to division by zero, accessing out of bound array indices, assignments of out of range values to variables, and so on. To ensure such problems do not occur, apply synthesis to the model. Alternatively, the CIF explorer may be used to verify that the specification does not contain runtime errors.
- It has bounded response. This property can be checked using the controller properties checker tool.
- It is confluent. This property can be checked using the controller properties checker tool.
- It is non-blocking under control. This property can be checked using the controller properties checker tool.

Pre-processing

To enlarge the accepted set of CIF files, the following transformations are first performed:

- Remove annotations (all annotations except documentation annotations are removed)
- Eliminate component definition/instantiation
- Eliminate state/event exclusion invariants
- Simplify values (if *Simplify values* option is enabled)
- Eliminate constants (if *Simplify values* option is enabled)
- Simplify others
- Remove I/O declarations

Specification checks

After pre-processing, a check is performed whether the CIF file meets all the requirements of the CIF PLC code generator:

- For components, the following conditions must be met:
  - No initialization predicates outside locations that restrict initialization.
  - No equations outside locations.
  - No events that can occur infinitely many times. The CIF PLC code generator only contains a partial check. A more complete check can be done by checking the bounded response property of the input specification with the controller properties checker tool.
- For automata and invariants, the following conditions must be met:
  - At least one automaton.
  - All automata must have a single initial location that can be statically decided.
  - Discrete variables must have a single initial value.
  - No state invariants that restrict behavior.
  - Only events that are either controllable or uncontrollable. This rejects events declared without `controllable` or `uncontrollable` keyword, edges with the `tau` event and edges without an event.
  - No urgent locations and edges.
  - No equations.
- For user-defined functions, the following conditions must be met:
  - No internal user-defined functions.
  - No external user-defined functions.
- For continuous variables the following conditions must be met:
  - Restricted continuous variables, see the continuous behavior section for details.
- For expressions and data types, the following conditions must be met:
  - No sets, dictionaries, strings, and stochastic distributions.
  - Only arrays, so no non-array lists.
  - Arrays must have at least one element, since `list[0]` cannot be expressed in a PLC system.
  - No nested arrays.
  - No arrays for the S7 targets.
  - For arrays: no `+`, `-`, `=`, `!=`, `empty`, `delete`, `in`, `pop`, `size`, list comprehensions and slicing.
  - For tuples: no `=` and `!=`.
  - Standard library functions: no `acosh`, `asinh`, `atanh`, `cbrt`, `cosh`, `pow`, `sinh`, and `tanh`.
  - No functions used as data values or stored in variables. Functions must thus only be used to call the function.
  - No use of variable `time`.

    - No `switch` expressions on array or tuple typed values (or parts of values).

  The restrictions on types and expressions do not apply to their use in values of documentation annotation arguments.

  Depending on the PLC target type, additional restrictions may apply. Please consult the next section for details.

- Furthermore:
    - I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
    - Annotations are ignored.

Target types

The PLC code generator output contains a single program called `MAIN`. This program is instantiated in a task, which is part of a resource, which in turn is part of a configuration. The configuration is put in a project, but that is outside the range of the IEC 61131-3 standard. Most implementations however, work with projects.

The effect of the project, configuration, resource, and task is dependent on the configured target. Unless overridden with options, the target also affects the shape of lower level details such as the form of the function calls, selected size of integers, and other configuration that tends to be different for different PLC systems. Last but not least, the target affects how the generated output gets written to files.

The PLC code generator supports the following targets:

- *ABB*:

  Code generation for ABB is currently experimental.

  Output for the ABB target generates multiple files in a directory. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_abb` directory extension is added. The `MAIN` program gets a `.plcprog` file extension, functions get a `.plcfunc` file extension, type declarations get a `.plctype` file extension, and the configuration gets a `.plccfg` file extension.

  The *PLC project name* and *PLC resource name* options have no effect for this target type.

- *IEC 61131-3*:

  When deploying code to a PLC, use a target specific to the PLC brand.

  Output for the IEC 61131-3 target generates multiple files in a directory. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_plc` directory extension is added. The `MAIN` program gets a `.plcprog` file extension, functions get a `.plcfunc` file extension, type declarations get a `.plctype` file extension, and the configuration gets a `.plccfg` file extension.

The *PLC project name* and *PLC resource name* options have no effect for this target type.

- *PLCopen XML*:

  When deploying code to a PLC, use a target specific to the PLC brand.

  PLCopen XML is an XML-based file format standardized by the PLCopen organization, intended for the exchange of complete projects of PLC code, even across different tools and vendors. The generated PLCopen XML file is compliant with version 2.01 of the PLCopen XML standard.

  Output for the PLCopen target generates a single file. By default, if no output file is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.plcopen.xml` file extension is added.

- *SIMATIC S7-300, S7-400, S7-1200, or S7-1500*:

  Code generation for S7-1200 and S7-1500 is currently experimental.

  Totally Integrated Automation Portal (TIA Portal) is an IDE for the development and testing of SIMATIC controllers. Using the S7 target, the generated PLC code can be imported in TIA Portal.

  Output for the SIMATIC target generates multiple files in a directory. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_s7_<ver>` (`<ver>` being either 1500, 1200, 400 or 300) directory extension is added. The `MAIN` programs and functions get a `.scl` file extension, type declarations get a `.udt` file extension, the persistent variables databases get a `.db` file extension, and the global variable lists get a `.xml` file extension.

  The *PLC task name*, *PLC task cycle time*, *PLC task priority*, *PLC project name*, *PLC configuration name*, and *PLC resource name* options have no effect for this target type. Currently, the PLC code generator cannot generate code for arrays for these systems. Therefore, all forms of CIF lists are forbidden for these target types.

  See the S7 PLC output section for details on how to transfer generated code to the PLC system.

- *TwinCAT*:

  TwinCAT is a complete IDE for the development and testing of PLC controllers. Using the TwinCAT PLC target type, the generated PLC code is written to the native file formats of the TwinCAT IDE.

  Output for the TwinCAT target generates multiple files in a directory tree. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_twincat` directory extension is added.

  See the TwinCAT PLC output section for details on how to transfer generated code to the PLC system.

I/O table file

The PLC code generator generates PLC code that reads input values from external inputs and writes output values to external outputs. Inside the PLC these values are transferred using variables with an I/O address. To make such connections in CIF, some of the input variables should be connected to input addresses, and some of the discrete and/or algebraic variables should be connected to output addresses. This is done by writing an I/O table file (details follow below), and giving it to the PLC code generator.

The generated PLC code will automatically read all inputs at the start of a PLC cycle and store them in the connected input variables. The values of the variables can for instance be used in a guard or a computation, or get copied to other variables. At the end of a cycle, outputs are written in a similar way by using the values of the connected discrete and algebraic variables.

The I/O table file is a **C**omma **S**eperated **V**alues (CSV) file. The CIF PLC code generator assumes the RFC-4180 format for this file, which is compatible with many spreadsheet programs. It is generally recommended to use a spreadsheet program to create and edit a CSV file.

Each line describes one connection with a variable. There are three or four columns in the CSV file. The first three columns are mandatory. The fourth column must either be completely omitted, or it must be added at all lines but entries can be empty. The information that should be provided in the columns is as follows:

- The first column contains the PLC input or output address.

  An example address is `%IX35.2.15`. It always starts with a `%`. The other parts are:

  - The `I` is the kind of connection. The `I` means it reads from an external input, a `Q` at that position means writing to an external output, and an `M` means reading or writing memory inside the PLC.
  - The `X` describes the amount of data being transferred. The `X` or not specifying a letter means 1 bit is transferred (in CIF, a `bool` type). A `B` at that position means transfer of a byte (8 bits), a `W` means transfer of a word (16 bits), a `D` means transfer of a double word (32 bits), and an `L` means transfer of a long word (64 bits).
  - The last part of an address is a sequence of numbers separated by dots. In the example it is `35.2.15`. The length and meaning of the sequence can vary between different PLC systems. Consult the manual of the target PLC system for details. In general the sequence indicates which of several available I/O ports is being used.
- The second column optionally contains the PLC type of the input or output. If left empty, the type of the CIF variable is used instead. In any case the fundamental type of the CIF variable must match with the fundamental type of the I/O port (both sides an integer, both sides a real, or both sides a boolean).
- The third column contains the non-escaped absolute name of the CIF variable connected to the I/O address. For example, the variable of the automaton below is named `G.A.v`:

  ```
  group G:
    automaton A:
      disc int v;
  ```

```
    ...
  end
end
```

- The fourth column optionally contains the name of the variable with the input or output address.

  If supplied, this name is used for the I/O variable declaration (in the PLC input/output variable table). Otherwise, the PLC code generator will generate a name from the CIF variable connected to the I/O address.

  Names must use the ASCII character set, and must be valid variable names for the used PLC target. The PLC code generator will automatically convert the name to the required form in the PLC code. For example, it may surround certain names with double quote characters.

  If at least one line specifies a name of the variable in the fourth column, then all lines must have a fourth column. An empty name is allowed for the lines where the default variable name suffices.

Depending on the PLC target type, additional restrictions may apply on addresses. Consult the manual of the target PLC system for details.

The kind of connection, the direction of the connection, and the allowed kind of variable are related in the following way:

Connection kind	Direction	Variable kind
`I`	from input	input variable
`Q`	to output	discrete or algebraic variable
`M`	from memory	input variable
`M`	to memory	discrete or algebraic variable

It is highly recommended that the number of transferred bits from the I/O matches with the CIF type of the variable as follows:

Transfer size	PLC type	CIF type
1	`BOOL`	`bool`
16	`INT`	`int` (for PLC systems that do not support 32 bit integers)
32	`DINT`	`int` (for PLC systems that support 32 bit integers)
32	`REAL`	`real` (for PLC systems that do not support 64 floating point)
64	`LINT`	`int` (for PLC systems that support 64 bit integers)
64	`LREAL`	`real` (for PLC systems that support 64 floating point)

If the size of the CIF variable type does not match with the size of the I/O port type, a cast expression is automatically inserted.

Note that the PLC code generator has no means to verify that the specified or derived type of the I/O port is compatible with the hardware device connected to the given I/O address.

Continuous behavior

Within PLC systems, a major use for continuous behavior is timers to detect (sufficient) passage of time. For instance, a reaction could be expected to happen within some period of time, but if it does not happen something else must be done. Another use case is to wait for a certain amount of time before a next action may be started. The PLC code generator assumes that continuous variables are always used for such purposes.

To reduce possible confusion with timer values, the common case of checking for timeout should be done against `0`. In that way, the check is always the same no matter what the initial timer value is. It is simpler to reason about positive values, so initial timer values should be positive. Finally, time is measured in seconds, since that is an SI unit, and a common amount of time for PLC controlled systems. This leads to a derivative of `-1` for the continuous variables. Unlike continuous variables in CIF, timers in the PLC stop measuring when they reach timeout. To preserve CIF behavior when generating PLC code, values used in comparisons against timers are restricted to non-negative values.

The following conditions therefore apply to continuous variables in the input CIF specification:

- The derivative of a continuous variable must always be `-1` or `-1.0`, and this value must be statically decidable.
- Continuous variables must only be compared to an inclusive upper-bound that is zero or larger. The upper-bound value must be statically decidable.

  For example, for a continuous variable `c`, the comparisons `c <= 0` or `5 >= c` could be used. The former example checks for timeout (zero seconds left), the latter example checks whether there are five seconds or less remaining.

- Each assignment of a new value to a continuous variable should be separate. That is, use `do c := 1.0, d := 2.0` on edges. Other forms like `do (c, d) := (1.0, 2.0)` are not allowed.
- Values used to initialize a continuous variable, or to assign to a continuous variable must be larger or equal to zero, and this must be statically decidable.

An example with a timer is shown in the next section.

Documenting PLC code

CIF has documentation annotations that can be attached to elements in a CIF specification. In such an annotation, design related information can be written about the element it is attached to. In that way, design information is available within the model, and it can be used, transported or converted by CIF applications.

The PLC code generator understands documentation annotations of specifications, groups,

automata, events, locations, edges, and discrete, continuous and input variables. Indirectly it also understand documentation annotations of invariants, since it uses CIF conversions to convert these to one of the previously mentioned CIF elements.

All documentation annotations that it understands in the input specification and that affect behavior of the system are copied into the generated PLC code as comments, where appropriate. For example, documentation annotations of the specification are only shown in the overall model overview. Documentation annotations of variables are shown whenever the variable is updated. Edge documentation annotations are shown when the edge is considered to be taken, and when it is actually taken.

Documentation annotations for CIF elements that are eliminated to other concepts or that are irrelevant are ignored. The former causes documentation of algebraic variables (these are inlined), constants, enumerations and their literals and type declarations to be ignored. Irrelevant CIF elements are for example empty automata, empty locations, or unused events. As these are not included in the PLC program, the documentation of such elements is also ignored.

As an example of how documentation gets added, consider the following CIF specification:

```
@doc("Enabled when the system is triggered to become active.")
uncontrollable reset;

@doc("Enabled when the system has been active long enough.")
uncontrollable timed_out;

@doc("Tracker to monitor activity of the system.")
plant ActivityTimer:
  @doc("Clock variable measuring remaining time until the system can be idle again.",
       "Currently configured to 10 seconds.")
  cont t der -1;

  @doc("System is idle.",
       "Waiting for the next reset.")
  location idle:
    initial;
    @@doc("Detected reset, jump into action.")
    edge reset do t := 10 goto active;

  @doc("System is active.")
  @doc("Should normally become idle again in 10 seconds.")
  location active:
    @@doc("Trigger detected while active, extend the active state.")
    edge reset do t := 10;

    @@doc("Time-out detected while active, go back to being idle.")
    edge timed_out when t <= 0 goto idle;
end
```

The example is extensively documented. Note that documentation annotations for the specification

and for the edges must have a double `@@`.

Below are a few examples of what can be generated in the PLC code. For all details, and to see the code generated for your specific target, please convert the example yourself and check what the PLC code generator produces.

A global overview description of the specification is created in the PLC code. It looks like:

```
(*-------------------------------------------------------
 * Model overview:
 *
 * ----
 * Automaton "ActivityTimer":
 * Tracker to monitor activity of the system.
 *
 * - Current location of automaton "ActivityTimer".
 * - Continuous variable "ActivityTimer.t".
 *   Clock variable measuring remaining time until the system can be idle again.
 *   Currently configured to 10 seconds.
 *
 * - PLC edge selection variable "edge_ActivityTimer_1".
 *
 * - Uncontrollable event "reset".
 *   Enabled when the system is triggered to become active.
 *
 * - Uncontrollable event "timed_out".
 *   Enabled when the system has been active long enough.
 *------------------------------------------------------- *)
```

PLC code that tests whether event `timed_out` can occur has a comment describing the CIF elements involved together with their documentation:

```
(***********
 * Test edge of automaton "ActivityTimer" to synchronize for event "timed_out".
 * This automaton must have an edge with a true guard to allow the event.
 *
 * Edge being tested:
 * - Location "active":
 *   System is active.
 *
 *   Should normally become idle again in 10 seconds.
 *
 *   - 2nd edge in the location
 *     Time-out detected while active, go back to being idle.
 ***********)
IF "DB".ActivityTimer = ActivityTimer_active AND "DB".ActivityTimer_t <=
DINT_TO_REAL(0) THEN
    edge_ActivityTimer_1 := 1;
ELSE
```

```
    (* The automaton has no edge with a true guard. Skip to the next event. *)
    eventEnabled := FALSE;
END_IF;
```

PLC Code that performs an edge (after deciding it should be taken) has a comment with the documentation of the edge:

```
ELSIF edge_ActivityTimer_1 = 2 THEN
    (***********
     * Perform assignments of the 1st edge in location "ActivityTimer.active".
     * Trigger detected while active, extend the active state.
     ***********)
    (* Perform update of continuous variable "ActivityTimer.t". *)
    "DB".ActivityTimer_t := 10.0;

    (* Reset timer of "ActivityTimer_t". *)
    ...  // Omitted for brevity.
END_IF;
```

Resulting PLC code

In general, the PLC code generator attempts to preserve as much (of the structure) of the original CIF specification as possible. For example, names of elements in CIF are also used for their converted equivalents in the generated PLC program. This makes it simpler to compare a CIF model against the generated code.

However, some parts of CIF specifications are not directly supported by any PLC system. For example, multiple automata and synchronizing on events or channels cannot be directly expressed in a PLC system. A PLC system does not have events, natively. For these cases the generator produces significantly different code, compared to the CIF specification. For example, for taking an event transition, all edges are grouped by event instead of grouping by automaton.

Other parts are supported only by some of the supported PLC systems. For these cases, the generator tries to generate the form that fits best within the abilities of the targeted PLC system and the configured options. For example, enumerations and constants are not always supported. The PLC code generator may change enumerations to constants or plain numbers in such a case.

From a bird's-eye view, the PLC code generator generates one main program that performs the entire PLC cycle. This main program adheres to the execution scheme defined by the CIF controller properties checker. Concretely, the main program consists of the following steps:

- Read values from the external inputs.
- For the first cycle, initialize the CIF variables and automata locations. For every next cycle, update the continuous variables for the passed time.
- Perform as many uncontrollable events as possible (within loop limits for uncontrollable events, see the *PLC maximum iterations* option for details).

- Perform as many controllable events as possible (within loop limits for controllable events, see the *PLC maximum iterations* option for details).
- Write values to the external outputs.

Performing as many uncontrollable (or controllable) events as possible is done as follows:

- A loop is entered. The loop tries to perform each uncontrollable (or controllable) event in turn.
- If none of the events was possible, the loop terminates.
- If the number of times that the loop was performed reaches the upper limit set by the *PLC maximum iterations* option for uncontrollable (or controllable) events, the loop terminates.
- Otherwise, the loop is executed again, looking for more uncontrollable (or controllable) events to perform.

The maximum number of iterations of a loop can be configured with the *PLC maximum iterations* option. Also explained there is the need to have a large enough maximum number of iterations in order to obtain the behavior as specified in the CIF model. If the specified maximum number of iterations is not `inf`, it may be difficult to judge whether the applied loop limits for controllable and uncontrollable event loops are large enough. The PLC code generator assists here by counting the number of times that a loop was exhausted, and storing this count in the `loopsExhausted` variable. The variable starts at `0` and increments by one each time an exhausted loop is detected, up to `MAX_LOOPS_EXHAUSTED` times (which is `9999`). By looking at the `loopsExhausted` variable in the running PLC, information can be obtained about the number of potentially prematurely aborted event loops since it started counting. If the value is `0`, everything is going well. A value larger than `0` indicates that in some PLC cycles more loop iterations were done than deemed possible at the time of generating the PLC code. Whether all necessary transitions were performed in those cases is not known, because the PLC code did not check further beyond the configured *PLC maximum iterations* limit.

Names

Generated PLC code contains names for PLC elements like functions, variables, enumeration literals, and so on. If such an element represents a CIF element in the PLC code, the absolute name of the CIF element is used as name in the PLC code. In this way it is easier to link parts of a CIF specification with parts of the generated PLC code.

However, name requirements in CIF are different from name requirements in a PLC system. As as result, names of CIF elements cannot always be used as-is in the PLC code, and must be changed before they can be used in the PLC code.

In general, names are preserved as much as possible. However, in case of confusion where a PLC name originates from, the PLC code generator has a *Rename warnings* option in its options. By enabling the option, the code generator will print warnings whenever a CIF name is not used as-is.

Note that the PLC code generator also generates output that is closely related to CIF elements, but not directly representing the CIF element itself. In such cases the variable may get a slightly different name to keep the name of the CIF element available for other uses, but at the same time

still show the connection with the CIF element.

Tests

To test the code generator, ESCET ships with a set of 'self-certification' tests. These tests can be used to generate code for specific targets and check that these execute correctly in the PLC development environment of that target.

The tests can be obtained from the ESCET IDE's welcome screen, by clicking the *Import CIF PLC code generator tests* button to start the import wizard for importing the tests. Alternatively, the wizard can be started from **File › New › Example...**, by selecting *CIF PLC code generator tests* under the *CIF* category.

Currently, these tests are only meant for testing the TwinCAT and S7 targets.

TwinCAT PLC output

TwinCAT is a complete IDE for the development and testing of PLC controllers. Using the TwinCAT PLC target type, the generated PLC code can be written to the native file formats of the TwinCAT IDE.

TwinCAT system modes

The TwinCAT software IDE is built on Microsoft Visual Studio. TwinCAT also supplies a utility to monitor and control the TwinCAT System Service, via an icon in the system tray, part in the Windows taskbar. The icon shows the mode (or status) of the TwinCAT system:

-  Inactive Mode
-  Stop Mode
-  Config Mode
-  Run Mode
-  Exception Mode

In the *Inactive Mode*, the TwinCAT System Service is not running. Restart Windows or manually activate the TwinCAT System Service and related services.

The *Stop Mode* is of little use, so switch to the *Config Mode* instead. In the *Config Mode*, the system (the PLC) can be configured. In the *Run Mode*, the system (the PLC) can be used to control the hardware. If a runtime exception occurs in *Run Mode*, the *Exception Mode* becomes active.

Within Visual Studio, the mode is also shown as an icon in the bottom right corner of the IDE:

-  Stop Mode
-  Config Mode
-  Run Mode
-  /  Free Run Mode (alternating icons)

-  Exception Mode

If the system is only equipped with a *soft PLC*, and the TwinCAT environment is set up to target the local system, both icons should indicate a consistent status. This page assumes this situation.

Changing modes

To change to *Config Mode* or restart *Config Mode* from the system tray:

- Right click the system tray icon and choose **System › Config**.

To change to *Config Mode* or restart *Config Mode* from Visual Studio:

- Click **TwinCAT › Restart TwinCAT (Config Mode)**. Alternatively, click the appropriate toolbar item ().
- A dialog pops up with the question *Restart TwinCAT System in Config Mode*. Click **[OK]** or press `Enter`.
- A dialog pops up with the question *Load I/O Devices*. Click **[Yes]** or press `Enter`.
- A dialog pops up with the question *Activate Free Run*. Click **[Yes]** or press `Enter`.

To change to *Run Mode* or restart *Run Mode* from the system tray:

- Make sure the hardware is powered up.
- Right click the system tray icon and choose **System › Start/Restart**.

To change to *Run Mode* or restart *Run Mode* from Visual Studio:

- Make sure the hardware is powered up.
- Click **TwinCAT › Restart TwinCAT System**. Alternatively, click the appropriate toolbar item ().
- A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.

In general it is recommended to switch modes via Visual Studio instead of via the system tray icon.

Generated output

The CIF PLC code generator can generate code into an existing TwinCAT Visual Studio solution, with a TwinCAT XAE project that contains a TwinCAT Standard PLC project. The directory that contains the solution must have the same name (except for the file extension) as the solution file (`.sln` file). The TwinCAT XAE project (both the directory and the `.tsproj` file) in the solution should have the same name as well. The TwinCAT XAE project should contain a TwinCAT Standard PLC project. This project may be given any name, and this name can be configured using the *PLC project name* option (*Generator* category). However, in order to be able to import I/O mappings it is highly

recommended to always use the default name suggested by TwinCAT (`Untitled1`) for TwinCAT Default PLC projects.

The project should already contain a task. This task can be given any name, and this name can be configured using the *PLC task name* option (*Generator* category). By default, TwinCAT Default PLC projects have a task named `PlcTask`. This name is the default for the *PLC task name* option as well. The cycle time and priority of the task are automatically updated by the PLC code generator, depending on the values provided for the options of the generator. TwinCAT code generation does not support disabling periodic task scheduling. The code generator also ensures that the `MAIN` program is instantiated/scheduled on the task.

All `POU` (functions, programs, etc), `DUT` (data types), and `GVL` (global variable lists) code files that are present in the TwinCAT Default PLC project are removed, from both the project and from disk, by the PLC code generator. The newly generated code files are (over)written, and the project is updated to include and compile only the newly generated files.

The *PLC configuration name* and *PLC resource name* options have no effect on the TwinCAT output.

Standard compliance

In general, the CIF PLC code generator follows the IEC 61131-3 PLC standard and does not use vendor-specific extensions. For the TwinCAT target however, the `STRUCT` data type declarations have no `;` character after the `STRUCT` data types, to work around a bug in the TwinCAT parser. In that case the output is not compliant with the IEC 61131-3 standard.

TwinCAT usage

Follow these steps to create a solution and project in TwinCAT, generate PLC code in it, and use the PLC code to control the hardware:

- Start TwinCAT (Visual Studio) by right clicking on the TwinCAT system tray icon and choosing **TwinCAT XAE (VS 2010)**.
- Create a new TwinCAT XAE project:
  - Click **File › New › Project...**.
  - In the *New Project* window, *TwinCAT XAE Project (XML format)* should already be selected. If not, first select *TwinCAT Project* on the left (from the list of *Installed Templates*), and then select *TwinCAT XAE Project (XML format)*.
  - Choose a project name and enter it in the *Name* box at the bottom of the dialog.
  - Make sure the *Solution name* is exactly the same, and that the *Create directory for solution* option is enabled.
  - For *Location*, select the directory in which the directory for the solution should be created. This is usually the directory that contains the script and/or source files you will be using to generate the PLC code.

        - To figure out the directory that contains the ToolDef script, make sure that the *Properties* view is visible in Eclipse. Select the ToolDef script in the *Project Explorer* tab or *Package Explorer* tab, and look for the *Location* property.
    - Click **[OK]** to create the project.
- Scan the available hardware:
    - Make sure the hardware is powered up.
    - Make sure the Config Mode is enabled, by switching to it if it is not the current mode.
    - In the project, under *I/O*, right click the *Devices* item and choose **Scan**. Alternatively, select the *I/O* item and click the appropriate toolbar item ().
    - A dialog pops up with the warning *HINT: Not all types of devices can be found automatically*. Click **[OK]** or press `Enter`.
    - A dialog pops up where you have to choose the I/O device to scan. Normally, if you have only one I/O device, it should already be selected. Click **[OK]** or press `Enter`.
    - A dialog pops up with the question *Scan for boxes*. Click **[Yes]** or press `Enter`.
    - A dialog pops up with the question *Activate Free Run*. Click **[Yes]** or press `Enter`.
- Create a new TwinCAT PLC project:
    - Right click the *PLC* item in the project, and choose **Add New Item...**.
    - In the *Add New Item* dialog, *Standard PLC project* should already be selected.
    - Keep the name `Untitled1` in the *Name* box at the bottom of the dialog.
    - Also don't change the *Location* of the project.
    - Click **[Add]** to add the PLC project.
- Save the entire solution by clicking **File › Save All**.
- Generate PLC code. If using a script, wait for the execution of the script to finish (the status indicated as part of the title of the console should change from `running` to `FINISHED`).
- Switch back to TwinCAT (Microsoft Visual Studio).
- When a *File Modification Detected* window pops up for the project, click **[Reload]** to reload the updated project that was written by the CIF tooling.
- Build the project by clicking **Build › Rebuild Solution**.
- Load the I/O mappings:
    - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
    - Select the mappings file for your hardware, and click **[Open]**.
    - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Activate the configuration:
    - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
    - A dialog pops up with the question *Activate Configuration (Old Configurations will be*

overwritten!). Click **[OK]** or press `Enter`.

- A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.

- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

To stop the PLC:

- Click **PLC › Stop**. Alternatively, click the appropriate toolbar item ().
- Click **PLC › Logout**. Alternatively, click the appropriate toolbar item (). This step is optional, and can usually be skipped.

To reset the PLC and start the execution from the beginning, with the same code and the same configuration:

- Make sure the hardware is powered up.
- Load the I/O mappings, if the *Untitled1 Instance - ...* item under the *Mappings* item is no longer present:
  - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
  - Select the mappings file for your hardware, and click **[Open]**.
  - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Reactivate the configuration:
  - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Activate Configuration (Old Configurations will be overwritten!)*. Click **[OK]** or press `Enter`.
  - A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.
- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

Whenever you change a CIF source file, regenerate a supervisor, change a hardware mapping, etc, you need to regenerate the PLC code as well. Follow these steps:

- Make sure the Config Mode is enabled, by switching to it if it is not the current mode.
- Save the entire solution in TwinCAT by clicking **File › Save All**.
- Regenerate PLC code from Eclipse. If using a script, wait for the execution of the script to finish (the status indicated as part of the title of the console should change from `running` to `FINISHED`).
- Switch back to TwinCAT (Microsoft Visual Studio).
- For all *File has been changed outside the environment. Reload the new file?* questions, click **[Yes]** or press `Enter`. See the Frequently Asked Questions (FAQ) for instructions on how to disable some of these questions.
- When a *File Modification Detected* window pops up for the project, click **[Reload]** to reload the updated project that was written by the CIF tooling.
- Clean the project by clicking **Build › Clean Solution**.
- Rebuild the project by clicking **Build › Rebuild Solution**.
- Make sure the hardware is powered up.
- Reload the I/O mappings:
  - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
  - Select the mappings file for your hardware, and click **[Open]**.
  - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Activate the configuration:
  - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Activate Configuration (Old Configurations will be overwritten!)*. Click **[OK]** or press `Enter`.
  - A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.
- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

Frequently Asked Questions (FAQ)

- **Question**: Is there something that can be done about the large number of *File has been changed*

outside the environment... prompts?

Answer: Yes. Click **Tools › Options...** to show the *Options* dialog. On the left, select **Environment › Documents**. Then, on the right, enable the *Detect when file is changed outside the environment* and *Auto-load changes, if saved* options.

- **Question**: Why doesn't importing the I/O mappings have any effect?

  Answer: Make sure the TwinCAT Standard PLC project is named `Untitled1`, and try again. Also, make sure you've built the solution before you try to import the I/O mappings.

- **Question**: I get a dialog with the warning *Device '...' needs a sync master (at least one variable linked to a task variable)*. What should I do?

  Answer: You probably forgot to load an I/O mapping.

- **Question**: Why does opening the MAIN program take so long in Run Mode?

  Answer: As the generated code becomes larger, it can take a long time for TwinCAT to set up the environment for inline monitoring of the MAIN program. To disable inline monitoring, click **Tools › Options...** to show the *Options* dialog. On the left, select **TwinCAT › PLC Environment › Text Editor**. Then, on the right, switch to the *Monitoring* tab. Disable the *Enable inline monitoring option*.

- **Question**: TwinCAT hangs for long periods of time, before crashing and automatically being restarted by Windows. What should I do?

  Answer: If the generated code is too large, TwinCAT may not be able to handle it. In such cases, reduce the size of the generated code, by reducing the size of the CIF model for which you are generating the code.

- **Question**: I get an error stating *Reading file failed. [...]*. What should I do?

  Answer: The generated code is too large for TwinCAT to handle. See also the answer to the previous question.

S7 PLC output

The S7 output (1500, 1200, 400 and 300) generates code for SIMATIC controllers. Totally Integrated Automation Portal (TIA Portal) is an IDE for the development and testing of SIMATIC controllers. Using the S7 output, the generated PLC code can be imported in TIA Portal. This allows for tight integration and smooth development cycles when implementing controllers on SIMATIC controllers.

Generated output

The CIF PLC code generator can generate code files that can be imported in TIA Portal (see S7 usage).

The following option restrictions apply:

- The *Convert enumerations* option must be set to *Automatically decide how to handle enumerations*, *Convert enumerations to integers* or *Convert enumerations to constants*.

The type of integer and real data types to use depends on the *PLC integer type size* and *PLC real type size* options. If *Automatic* is selected, the following data types are used:

S7 type	Integer type	Real type
S7-1500	`DINT` (32 bit)	`LREAL` (64 bit)
S7-1200	`DINT` (32 bit)	`LREAL` (64 bit)
S7-400	`DINT` (32 bit)	`REAL` (32 bit)
S7-300	`DINT` (32 bit)	`REAL` (32 bit)

The *PLC task name*, *PLC task cycle time*, *PLC task priority*, *PLC project name*, *PLC configuration name*, and *PLC resource name* options have no effect for this output type.

Standard compliance

In general, the CIF PLC code generator follows the IEC 61131-3 PLC standard and does not use vendor-specific extensions. For the Siemens S7 targets however, there are some differences. For instance, the persistent variables are stored in a separate database file, named `DB`, and all references to persistent variables are prefixed with `"DB".`. Also, global variables and constants are stored in a tag table, instead of a global variable list. In such cases the output is not compliant with the IEC 61131-3 standard.

S7 usage

Below, the basic usage of generated S7 code with TIA Portal is explained. The steps assume that TIA Portal v15 is used. If you use a different version of TIA Portal, the steps may be slightly different.

Follow these steps to create a project in TIA Portal, import PLC code in it, and use the PLC code to control the hardware:

- Start TIA Portal.
- Create a new S7 project:
  - In the *Portal view*, click **Create new project**.
  - In the *Create new project* window, choose a project name and enter it in the *Project name* box.
  - For *Path*, select the directory in which the directory for the project should be created.
  - Click **[Create]** to create the project.
  - Switch to the *Project view*.

- Add the PLC device:
  - Unfold the project and double click *Add new device*.
  - Unfold the device type category (that is, S7-1500, S7-1200, S7-400 or S7-300).
  - Select the device with the correct article number.
  - Click **[OK]**.
- Import the generated PLC code (PLC tags and constants):
  - Double click *Default tag table* inside the *PLC tags* folder.
  - For each generated `.xml` file (e.g., *CONSTANTS.xml*, *INPUTS.xml*, *OUTPUTS.xml*), do the following:
    - Click the *Import* button.
    - Select the file.
    - Make sure *Tags* and *Constants* are both selected.
    - Click **[OK]**.
- Import the generated PLC code (Program blocks):
  - Unfold the *External source files* folder inside the device folder.
  - Double click *Add new external files*.
  - Select all generated `.scl`, `.udt` and `.db` files, by selecting the first (top) file, and then selecting the last (bottom) file while holding the `Shift` key.
  - Click the **[Open]** button.
  - Select all imported files (using `Shift` as above), right-click one of them and select **Generate blocks from source**.
    - If asked whether it is OK that *Blocks can be overwritten*, click the **[OK]** button.
    - Ignore any warnings or errors.
- Compile the project:
  - Right-click *Program blocks* and select **Compile › Software (rebuild all)**.
  - If there are warnings or errors:
    - Compile again.
    - If any compile errors remain, repeat the steps above.
    - If errors about timers remain:
      - Sometimes, timers are not imported correctly. In that case, compilation fails. To add the timers manually, do the following.
      - Open *MAIN* inside the *Program blocks* folder.
      - Type `TON` and click `Enter`. The *Call options* window pops up.
      - In the *Call options* window, for each timer, enter its name in the *Name* box and click **[OK]**.
      - Delete the generated lines of code.

        - Compile the project again.
    - If errors about tuples remain, the program blocks need to be created in the right order, based on tuple dependencies:
        - Generate the program blocks for the structs of the tuple types from the `.udt` files first, and create program blocks for the `.scl` and `.db` files last.
        - Generate the program blocks for the tuples in the right order. If one `.udt` file uses a type from another tuple struct of another `.udt` file, the program block for the former needs to be created after the program block for the latter. The PLC code generator writes a text file `derived_type_dependencies.txt` that contains the dependency information between structs, and from structs to enumerations if the latter types are used in a struct.
- Download PLC code to device:
    - Click the *Download to device* icon in the task bar. The *Extended download to device* window pops up.
    - Click **[Start search]**.
    - When a device has been found, click **[Load]**. The *Load preview* window pops up.
    - Depending on the state of the PLC (idle, running, runtime error, ...), TIA Portal may require you to take certain actions. Consult the TIA Portal documentation for further information.
    - Click **[Load]**.
    - Change the *Start modules* action from *No action* to *Start module*.
    - Click **[Finish]**.

If a (simulated) PLC indicates a System Fault (SF), try to alter the memory allocation properties (may only apply to S7-400):

- Open the *Device Configuration* for that PLC.
- Select the *CPU*, e.g. *CPU_1*, by clicking on it.
- Click the *Properties* button to show the properties of the selected CPU.
- On the *General* tab, select the *Memory* category.
- Increase the *Number of bytes* property for *Priority class* number *1*.
- If needed, also increase the *Bytes of max*.

To debug the PLC code:

- Click *Go online* in the task bar.
- The values of various variables can now be inspected.
    - The values of input and output variables can be seen by opening *Default tag table* inside the *PLC tags* folder. Click the *Monitor all* icon. The actual values appear in the *Monitor value* column.
    - The values of state variables can be seen by opening *DB* under *Program blocks*. Click the *Monitor all* icon. Unfold the *Static* item. The actual values appear in the *Monitor value*

column.

To stop the PLC:

- While online, click **[STOP]** in the *CPU operation panel* in the *Testing* tab.
- Click *Go offline* in the task bar.

To reset the PLC and start from the initial state, with the same code and the same configuration:

- While online, click **[MRES]** in the *CPU operation panel* in the *Testing* tab. The *Online & diagnostics* window appears.
- Click **[Yes]**.
- If the PLC was in run mode, a new *Online & diagnostics* window appears. Click **[Yes]**.
- Click *Go online* in the task bar.
- Start the PLC by clicking **[RUN]** in the *CPU operation panel* in the *Testing* tab.
- The PLC is now running, and you can use the system.

4.5.3. CIF PLC code generator (deprecated)



This application is **deprecated**, and is no longer being actively developed. Use the new CIF PLC code generator instead.

The CIF PLC code generator can be used to generate **P**rogrammable **L**ogic **C**ontroller (PLC) code, for the implementation of CIF controllers on PLCs. The generated PLC code adheres to the IEC 61131-3 standard, a widely used international standard for PLC programming languages.

Starting the generator

The generator can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF implementation tools › Generate PLC code for CIF specification... (deprecated)**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF implementation tools › Generate PLC code for CIF specification... (deprecated)**.
- Use the `cif2plc` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2plc` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification.
- *Output file*: The absolute or relative local file system path to the output file. Depending on the output type, this is either a file or a directory. The default depends on the output type as well.
- *PLC code output type*: The PLC code output type specifies the type of output (essentially the file format) that should be generated.
- *PLC task name*: The name of the PLC task to generate. If not specified, defaults to `PlcTask`.
- *PLC task cycle time*: If periodic task scheduling is to be used, use this option to enable it, by setting the cycle time in milliseconds for the task to generate. The cycle time must be a positive integer number. Cycle times of over a minute may not work as expected. However, since such large cycle times are of little practical use, that should not be a problem. Disable this option to disable periodic task scheduling. This option defaults to periodic task scheduling with a cycle time of 10 milliseconds. Disabling periodic task scheduling is not supported for the TwinCAT output type.
- *PLC task priority*: The scheduling priority of the task to generate. The priority must be in the range [0 .. 65535], where priority 0 is the highest priority, and priority 65535 is the lowest priority. The priority can be used for preemptive or non-preemptive scheduling. If not specified, the priority defaults to 20.
- *PLC project name*: The name of the PLC project to generate. If not specified, defaults to `Untitled1`.
- *PLC configuration name*: The name of the PLC configuration to generate. If not specified, defaults to `Untitled1`.
- *PLC resource name*: The name of the PLC resource to generate. If not specified, defaults to `Untitled1`.
- *PLC number bits*: The maximum number of bits supported by the PLC for representing numeric values. Use 64-bit if supported by the PLC, for full CIF compatibility, and the highest accuracy. If the PLC does not supported 64-bit integer and floating point numbers, set this option to 32-bit. Use the automatic option to automatically select the maximum number of bits depending on the output type. For the PLCopen XML, IEC 61131-3 and TwinCAT output types, the automatic option is 64-bits. For the S7 output type, see the table on the S7 page.
- *PLC maximum iterations*: The maximum number of iterations of the main loop of the main program body, per execution of the main program body. Must be a positive number. May be set to infinite, to not impose a restriction. The default is set to at most 100 iterations. See the section on the program body for more information.
- *Formal function invocation (arguments based)*: Specifies for which functions to use formal invocation syntax in the generated PLC code. By default the formal syntax is used for none of the functions. For PLC implementations that don't support the non-formal syntax, the formal syntax can be used for all functions, or only for functions with more than one argument.
- *Formal function invocation (function kind based)*: Specifies for which functions to use formal invocation syntax in the generated PLC code. By default the formal syntax is used only for all but the standard library/conversion functions. This can be changed to only the standard library/conversion functions, or to all functions.
- *Convert enumerations*: Specifies whether enumerations should be converted before generating the actual PLC code, in case the PLC doesn't support enumerations. Enumerations can be

converted to integers or constants. By default enumerations are not converted.

- *Simplify values*: Enable this option to simplify values and inline constants during generation of the PLC code. The benefit is that static computations are evaluated, and for the result simpler PLC code is generated. This applies to both the original model as well as anything generated during the code generation, for instance as the result of linearization. Constants created by *Convert enumerations* are not inlined. Enabling the option may thus lead to simpler PLC code. However, the simplification itself may take quite some time. Furthermore, as constants are inlined, they are duplicated for each use. If a constant with a large value (such as a large literal array) is used multiple times, this can lead to significantly larger PLC code. By default this option is enabled.
- *Rename warnings*: Enable this option to print warnings to the console when a PLC name is renamed due to a conflict with another PLC name, or disable it to omit the warnings. The option is disabled by default. See the section on names for more information.

Formal function invocation syntax (e.g. `MIN(IN1:3, IN2:5)`) is used in the generated PLC code only for those functions that are included by both the *Formal function invocation (arguments based)* option and the *Formal function invocation (function kind based)* option. If either of the options excludes the function, the non-formal function invocation syntax (e.g. `MIN(3, 5)`) is used for that function. Note that for invocations of function blocks, the generated PLC code always uses the formal syntax. Furthermore, for certain standard library functions, only the non-formal syntax is defined by the standard, and thus used in the generated PLC code.

Supported specifications

The CIF PLC code generator supports a subset of CIF specifications. The following restrictions apply:

- Unrestricted events are not supported, as they can always be executed (are always enabled), and would lead to infinitely running PLC code. Detection of unrestricted events however, is incomplete. That is, for some unrestricted the code generation results in an error, while for other unrestricted events code is generated without any warnings or errors. See the Program body section for more information.
- Specifications without automata are not supported.
- Initialization predicates outside locations are not supported, except if it can be determined statically that they are trivially `true`.
- State invariants (in components as well as locations) are not supported, except if it can be determined statically that they are trivially `true`.
- Discrete variables with multiple initial values (including `any`) are not supported.
- External user-defined functions are not supported.
- Urgent locations are not supported.
- Initialization predicates in locations that can not be statically evaluated are not supported.
- Automata that do not have exactly one initial location are not supported.
- Urgent edges are not supported.

- User-defined functions without parameters are not supported.
- Internal user-defined functions that contain a multi-assignment that assigns multiple (non-overlapping) parts of a single variable are not supported. For instance, `a[0], a[1] := 1, 2` is not supported, as multiple parts (the first and second element) of array `a` are assigned in a single multi-assignment. This restriction applies only to multi-assignments in functions, not to multi-assignments on edges.
- Internal user-defined functions that contain a `continue` statement are not supported.
- Data types other than `bool`, `int` (with or without range), `real`, enumerations, tuples, and arrays, are not supported. This applies to the data types of variables, parameters of functions, return types of functions, etc. For the S7 output type, arrays are not supported.
- String values are not supported.
- Cast from `int` to `real` and casts that don't change the type, are supported. All other casts are not supported.
- Sampling of distributions is not supported.
- The use of the conjunction and disjunction binary operators on anything other than boolean values is not supported.
- The use of the equality and inequality binary operators on anything other than boolean, integer, real, or enumeration values is not supported.
- The use of the addition and subtraction binary operators on anything other than integer or real values is not supported.
- The use of the element test and subset binary operators is not supported.
- Projection on anything other than tuples and arrays is not supported. This applies to expressions as well as addressables (the left hand sides of assignments). For arrays, both 0-based indices (counting from the left) as well as negative indices (counting from the right) are supported.
- List comprehensions and slicing are not supported.
- Function calls on anything other than standard library functions and internal user-defined functions is not supported.
- The `ceil`, `del`, `empty`, `floor`, `fmt`, `pop`, `round`, `scale`, `sign`, `size`, `acosh`, `asinh`, `atanh`, `cosh`, `sinh`, and `tanh` standard library functions are not supported.
- The distribution standard library functions are not supported.
- Lists (except for arrays), sets, and dictionaries are not supported. For the S7 output type, arrays are not supported.
- The use of functions as values is not supported. That is, functions may only be used in function calls, and may for instance not be stored in variables.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
- Annotations are ignored.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove annotations
- Eliminate component definition/instantiation
- Eliminate state/event exclusion invariants
- Simplify values (if *Simplify values* option is enabled)
- Eliminate constants (if *Simplify values* option is enabled)
- Simplify others
- Remove I/O declarations

After these preprocessing steps, the generator checks whether the specification is supported. It then applies the following additional CIF to CIF transformations:

- Linearize (merge)
- Merge enumerations
- Simplify values (if *Simplify values* option is enabled)
- Add default initial values
- Convert enumerations to integers (if *Convert enumerations* option is set to integers)
- Convert enumerations to constants (if *Convert enumerations* option is set to constants)

Standard compliance

The PLC code that is generated complies with the second edition (from 2003) of the IEC 61131-3 standard. The code does not use any vendor specific extensions, functions, etc.

For the TwinCAT output type, `STRUCT` data type declarations have no `;` character after the `STRUCT` data types, to work around a bug in the TwinCAT parser. In that case the output is not compliant with the IEC 61131-3 standard.

For the S7 output type, persistent variables are stored in a separate database file, named `DB`. All references to persistent variables are prefixed with `"DB".`. Global variables and constants are stored in a tag table, instead of a global variable list. In those cases the output is not compliant with the IEC 61131-3 standard.

Generated PLC code

The bodies of the programs and functions in the generated PLC code use Structured Text as programming language. Structured Text is one of the programming languages that is part of the IEC 61131-1 standard.

The generated code contains a single program called `MAIN`. This program is instantiated in a task, which is part of a resource, which in turn is part of a configuration. The configuration is put in a project, but that is outside the range of the IEC 61131-3 standard. Most implementations however, work with projects. The effect of the project, configuration, resource, and task is dependent on the output type. Besides a program, the code generator also generates data types, functions, and global variables.

For algebraic variables, functions are generated that are parametrized over the state of the system. Similarly, functions are generated for the derivatives of the continuous variables. For the discrete and continuous variables (including variable `time`), a `STRUCT` data type named `STATE` is generated that holds the entire state of the system (or more precisely, the state of the CIF specification). For input variables, an `INPUTS` global variables list/file is generated. For constants (if *Simplify values* option is disabled), a `CONSTS` global variables list/file is generated.

For invalid CIF specifications, the resulting code may not always behave as the original CIF code. For instance, the ranges of integer types are ignored by the code generator. Furthermore, `mod` by zero in CIF results in a runtime error, while in the generated code it results in zero. Out of bound projection indices of arrays may for instance have the effect that the closest existing element is used instead, and the result may even be implementation dependent. There are other such differences as well.

Little to no effort is put into generating readable code. That is, the generated is not meant to be inspected. The generated code is also not intended to be modified directly. Instead, the original CIF models should be modified, and from them new PLC code should be generated. Examples of non-readable code include the generated names, and code generated for expressions (which has way more parentheses than strictly necessary).

Names

The generated PLC code will contain names for functions, variables, locations of automata, etc. The absolute names of all objects are used. The names are influenced by the linearization algorithm. They are further influenced by the PLC code generator.

The code generator ensures unique names in the generated PLC code, taking into account the reserved names (keywords, standard functions, etc) of the IEC 61131-3 standard and some of the common implementations of that standard. To avoid reserved names, names are prefixed with prefixes that depend on the type of object (for instance, `dvar_` for discrete variables). Furthermore, it takes into account that names in CIF are case sensitive, while they are case insensitive in IEC 61131-3. The code generator computes a candidate name based on these criteria.

If this results in duplicate names, renaming is performed. A common reason for renaming is different locations (of the same automaton or of different automata) that have the same name, but in different casing. For instance, consider one location named `WaitForTimeout` and another named `WaitForTimeOut`. After linearization, there is an enumeration with a literal for each unique location name. The enumeration thus has at least two literals, named `WaitForTimeout` and `WaitForTimeOut`. The candidate names for these enumeration literals are `elit_WaitForTimeout` and `elit_WaitForTimeOut`. However, they are identical except for casing. Thus, the first one that is

encountered during code generation gets the candidate name, and the second one is renamed. This results in a warning like this:

```
WARNING: PLC variable "elit_WaitForTimeout" is renamed to "elit_WaitForTimeout2".
```

Renaming may be needed to ensure valid functioning output. However, it may make it slightly more difficult to trace names in the generated PLC code back to the original CIF model. Also, in the PLC environment, when debugging the current values of variables may be shown. If enumeration literals are renamed, the renamed names may then be shown in the debugger. By default, the PLC code generator does not inform the user when renaming takes place. By enabling the *Rename Warnings* option, a warning will be printed to the console, each time a PLC name is renamed.

Program body

The body of the `MAIN` program consists of several parts. The `MAIN` program is executed exactly once per cycle, by the PLC. The following parts are present, and are executed in order:

- Time and cycle time handling.
- Initialization of the initial state, for the first cycle only.
- Updating of the continuous variables for the time that has passed since the previous cycle, for all but the first cycle.
- A loop that executes events for as long as they are possible.

The code automatically determines the cycle time that is used, in a manner that does not require any vendor specific extensions, functions, or data types. The generated code works for fixed cycle times (periodic task scheduling) as well as variable cycle times (non-periodic tasks).

During the first cycle, the variables of the state are initialized to their proper values. For subsequent cycles, the state resulting from the previous cycle is reused, where the values of the continuous variables are updated to account for the time that has passed since the beginning of the previous cycle. For details on the method used to update the values of the continuous variables, see the Accuracy over time section.

The loop at the end of the body evaluates the guard of each of the events of the CIF specification. If a guard holds, the corresponding update is performed. Code is generated for the events, or rather the edges, in the order that they occur in the linearized CIF specification. Linearization is performed in such a way, that the PLC code that is generated from it, has a correspondence to the simulator. Assuming the simulator is used to simulate by always automatically choosing the first transition that is enabled, the PLC code will choose that same transition. That is, for each iteration of the event loop, the code is executed for the first event with an enabled guard. Then, a new iteration of the loop is started, from the top, to ensure that the first enabled event is always executed, similar to simulation. If during an iteration of the loop none of the events is enabled, the loop terminates.

If at least one event was enabled, the loop is executed again. If during an iteration of the loop none

of the edges was enabled, the loop terminates. This approach generally does not result in predictable execution times of the program body. That is, if a certain edge keeps being enabled, the loop is executed over an over again. In that case, the execution time of the body exceeds the fixed cycle time, and the controller is no longer guaranteed to work correctly. Even if the loop is only executed a finite number of times, the execution time of the body may exceed the fixed cycle time. One may consider monitoring the execution times to detect such issues.

Within the 'event loop', a counter is used to detect how often the loop is repeated. After the loop is executed a 100 times, the loop is terminated, to ensure that the body terminates in the case of events that are always possible. In such a cases, variable `loopsKilled` is incremented by one, for debugging. While by default the loop is terminated after 100 iterations, this maximum number of iterations can be configured using the *PLC maximum iterations* option (*Generator* category). The option can also be set to infinite, to not impose a restriction, and never terminate the loop due to too many iterations. See also the Options section above.

Obviously, similar to the fact that the controller cannot guarantee its correct behavior if the execution time of a cycle exceeds the cycle time, the controller also cannot guarantee its correct behavior if the inputs from the external I/O change more rapidly than the cycle time. That is, if an input changes during a cycle, the change is not noticed until the next cycle. If an input changes during a cycle and changes back during that same cycle, the change is not noticed at all. Therefore, choosing an appropriate cycle time is essential for the correct functioning of the controller.

Accuracy over time

The generated code maintains the running time in seconds, by adding the cycle time to current running time, on each cycle. Due to the finite binary representation of real values, this results in loss of a accuracy over prolonged periods of execution time. As such, the use of variable `time` in CIF models is highly discouraged.

However, for a cycle time of one millisecond, while there may be loss of accuracy, time should keep increasing for at least several thousand years. After that, adding the cycle time may no longer increase the value of the variable, potentially resulting in unexpected execution behavior.

The continuous variables from the CIF specification are updated each cycle using the Euler method for integration. That is, each cycle the cycle time is multiplied by the derivative of the continuous variable in the state of the previous cycle. This value is added to the value of the continuous variable. Essentially, a linear approximation of the derivative is used, calculated using the state of the previous cycle.

The Euler method can be numerically unstable, can suffer from rounding errors due to the use finite binary representations of real values, and has some other issues as well. The effects are likely to be limited if linear continuous variable (continuous variables with constant derivatives) are used. Especially clocks (derivative +1 or -1) generally don’t suffer too much from these issues. However, even clocks may suffer from loss of accuracy over time. As such, clocks should not be used to measure over long periods of time. Instead they should be reset, rendering the past loss of accuracy irrelevant for any future measurements.

I/O coupling

For all variables in the state of the CIF specification, except for variable `time`, PLC variables are generated with a `%Q*` address, to ensure they can be used as output variables, and can potentially be coupled to output ports. The state is maintained in a variable `state0`, a persistent local variable of the `MAIN` program. For the S7 output type, no output addresses are generated. Instead, new output variables should be declared in a tag table. Values can be assigned to the new output variables at the end of the main program.

For all input variables of the CIF specification, a PLC variable is generated with a `%I*` addresses, to ensure they can be used as input variables, and can potentially be coupled to input ports. These variables are generated in a global variables list/file named `INPUTS`.

The actual coupling to hardware addresses is something that is left to the implementation of the PLC programming environment.

PLC output type

The PLC code generator can generate its output in multiple forms (file formats) depending on the output type. The output type can be configured using the *PLC code output type* option (*Generator* category). The following output types are currently available:

- PLCopen XML (default)
- IEC 61131-3
- TwinCAT
- S7 (1500, 1200, 400 and 300)

PLCopen XML is an XML-based file format standardized by the PLCopen organization, intended for the exchange of complete projects of PLC code, even across different tools and vendors. The PLCopen XML files generated by the PLC code generator are compliant with version 2.01 of the PLCopen XML standard. By default, if no output file is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.plcopen.xml` file extension is added.

Output in IEC 61131-3 syntax generates multiple files. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_plc` directory extension is added. The `MAIN` program gets a `.plcprog` file extension, functions get a `.plcfunc` file extension, type declarations get a `.plctype` file extension, and the configuration gets a `.plccfg` file extension. The *PLC project name* and *PLC resource name* options have no effect for this output type.

TwinCAT is a complete IDE for the development and testing of PLC controllers. Using the TwinCAT PLC output type, the generated PLC code can be written to the native file formats of the TwinCAT IDE. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_twincat` directory extension is added. See the TwinCAT PLC output page for further details.

The S7 output (1500, 1200, 400 and 300) generates code for SIMATIC controllers. Totally Integrated

Automation Portal (TIA Portal) is an IDE for the development and testing of SIMATIC controllers. Using the S7 output, the generated PLC code can be imported in TIA Portal. By default, if no output directory is specified, it defaults to the input file path, where the `.cif` file extension is removed (if present), and a `_s7_<ver>` (`<ver>` being either 1500, 1200, 400 or 300) directory extension is added. Output for S7 generates multiple files. The `MAIN` programs and functions get a `.scl` file extension, type declarations get a `.udt` file extension, the persistent variables databases get a `.db` file extension, and the global variable lists get a `.xml` extension. The *PLC task name*, *PLC task cycle time*, *PLC task priority*, *PLC project name*, *PLC configuration name*, and *PLC resource name* options have no effect for this output type. See the S7 PLC output page for further details.

TwinCAT PLC output

TwinCAT is a complete IDE for the development and testing of PLC controllers. Using the TwinCAT PLC output type, the generated PLC code can be written to the native file formats of the TwinCAT IDE. This allows for more tight integration and smoother development cycles than using the PLCopen XML output type.

We currently only provide official support for version 3.1 of TwinCAT.

TwinCAT system modes

The TwinCAT software IDE is built on Microsoft Visual Studio. TwinCAT also supplies a utility to monitor and control the TwinCAT System Service, via an icon in the system tray, part in the Windows taskbar. The icon shows the mode (or status) of the TwinCAT system:

- Inactive Mode
- Stop Mode
- Config Mode
- Run Mode
- Exception Mode

In the *Inactive Mode*, the TwinCAT System Service is not running. Restart Windows or manually activate the TwinCAT System Service and related services.

The *Stop Mode* is of little use, so switch to the *Config Mode* instead. In the *Config Mode*, the system (the PLC) can be configured. In the *Run Mode*, the system (the PLC) can be used to control the hardware. If a runtime exception occurs in *Run Mode*, the *Exception Mode* becomes active.

Within Visual Studio, the mode is also shown as an icon in the bottom right corner of the IDE:

- Stop Mode
- Config Mode
- Run Mode
- / Free Run Mode (alternating icons)
- Exception Mode

If the system is only equipped with a *soft PLC*, and the TwinCAT environment is set up to target the local system, both icons should indicate a consistent status. This page assumes this situation.

Changing modes

To change to *Config Mode* or restart *Config Mode* from the system tray:

- Right click the system tray icon and choose **System › Config**.

To change to *Config Mode* or restart *Config Mode* from Visual Studio:

- Click **TwinCAT › Restart TwinCAT (Config Mode)**. Alternatively, click the appropriate toolbar item ().
- A dialog pops up with the question *Restart TwinCAT System in Config Mode*. Click **[OK]** or press `Enter`.
- A dialog pops up with the question *Load I/O Devices*. Click **[Yes]** or press `Enter`.
- A dialog pops up with the question *Activate Free Run*. Click **[Yes]** or press `Enter`.

To change to *Run Mode* or restart *Run Mode* from the system tray:

- Make sure the hardware is powered up.
- Right click the system tray icon and choose **System › Start/Restart**.

To change to *Run Mode* or restart *Run Mode* from Visual Studio:

- Make sure the hardware is powered up.
- Click **TwinCAT › Restart TwinCAT System**. Alternatively, click the appropriate toolbar item ().
- A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.

In general it is recommended to switch modes via Visual Studio instead of via the system tray icon.

Generated output

The CIF PLC code generator can generate code into an existing TwinCAT Visual Studio solution, with a TwinCAT XAE project that contains a TwinCAT Standard PLC project. The directory that contains the solution must have the same name (except for the file extension) as the solution file (`.sln` file). The TwinCAT XAE project (both the directory and the `.tsproj` file) in the solution should have the same name as well. The TwinCAT XAE project should contain a TwinCAT Standard PLC project. This project may be given any name, and this name can be configured using the *PLC project name* option (*Generator* category). However, in order to be able to import I/O mappings it is highly recommended to always use the default name suggested by TwinCAT (`Untitled1`) for TwinCAT Default PLC projects.

The project should already contain a task. This task can be given any name, and this name can be configured using the *PLC task name* option (*Generator* category). By default, TwinCAT Default PLC projects have a task named `PlcTask`. This name is the default for the *PLC task name* option as well. The cycle time and priority of the task are automatically updated by the PLC code generator, depending on the values provided for the options of the generator. The code generator also ensures that the `MAIN` program is instantiated/scheduled on the task.

All `POU` (functions, programs, etc), `DUT` (data types), and `GVL` (global variable lists) code files that are present in the TwinCAT Default PLC project are removed, from both the project and from disk, by the PLC code generator. The newly generated code files are (over)written, and the project is updated to include and compile only the newly generated files.

The *PLC configuration name* and *PLC resource name* options have no effect on the TwinCAT output.

TwinCAT usage

Follow these steps to create a solution and project in TwinCAT, generate PLC code in it, and use the PLC code to control the hardware:

- Start TwinCAT (Visual Studio) by right clicking on the TwinCAT system tray icon and choosing **TwinCAT XAE (VS 2010)**.
- Create a new TwinCAT XAE project:
  - Click **File › New › Project...**.
  - In the *New Project* window, *TwinCAT XAE Project (XML format)* should already be selected. If not, first select *TwinCAT Project* on the left (from the list of *Installed Templates*), and then select *TwinCAT XAE Project (XML format)*.
  - Choose a project name and enter it in the *Name* box at the bottom of the dialog.
  - Make sure the *Solution name* is exactly the same, and that the *Create directory for solution* option is enabled.
  - For *Location*, select the directory in which the directory for the solution should be created. This is usually the directory that contains the script and/or source files you will be using to generate the PLC code.
    - To figure out the directory that contains the ToolDef script, make sure that the *Properties* view is visible in Eclipse. Select the ToolDef script in the *Project Explorer* tab or *Package Explorer* tab, and look for the *Location* property.
  - Click **[OK]** to create the project.
- Scan the available hardware:
  - Make sure the hardware is powered up.
  - Make sure the Config Mode is enabled, by switching to it if it is not the current mode.
  - In the project, under *I/O*, right click the *Devices* item and choose **Scan**. Alternatively, select the *I/O* item and click the appropriate toolbar item ().
  - A dialog pops up with the warning *HINT: Not all types of devices can be found automatically*.

Click **[OK]** or press `Enter`.

- A dialog pops up where you have to choose the I/O device to scan. Normally, if you have only one I/O device, it should already be selected. Click **[OK]** or press `Enter`.
- A dialog pops up with the question *Scan for boxes*. Click **[Yes]** or press `Enter`.
- A dialog pops up with the question *Activate Free Run*. Click **[Yes]** or press `Enter`.

- Create a new TwinCAT PLC project:
  - Right click the *PLC* item in the project, and choose **Add New Item...**.
  - In the *Add New Item* dialog, *Standard PLC project* should already be selected.
  - Keep the name `Untitled1` in the *Name* box at the bottom of the dialog.
  - Also don't change the *Location* of the project.
  - Click **[Add]** to add the PLC project.
- Save the entire solution by clicking **File › Save All**.
- Generate PLC code. If using a script, wait for the execution of the script to finish (the status indicated as part of the title of the console should change from `running` to `FINISHED`).
- Switch back to TwinCAT (Microsoft Visual Studio).
- When a *File Modification Detected* window pops up for the project, click **[Reload]** to reload the updated project that was written by the CIF tooling.
- Build the project by clicking **Build › Rebuild Solution**.
- Load the I/O mappings:
  - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
  - Select the mappings file for your hardware, and click **[Open]**.
  - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Activate the configuration:
  - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Activate Configuration (Old Configurations will be overwritten!)*. Click **[OK]** or press `Enter`.
  - A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.
- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

To stop the PLC:

- Click **PLC › Stop**. Alternatively, click the appropriate toolbar item ().
- Click **PLC › Logout**. Alternatively, click the appropriate toolbar item (). This step is optional, and can usually be skipped.

To reset the PLC and start the 'simulation' from the beginning, with the same code and the same configuration:

- Make sure the hardware is powered up.
- Load the I/O mappings, if the *Untitled1 Instance - ...* item under the *Mappings* item is no longer present:
  - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
  - Select the mappings file for your hardware, and click **[Open]**.
  - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Reactivate the configuration:
  - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Activate Configuration (Old Configurations will be overwritten!)*. Click **[OK]** or press `Enter`.
  - A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.
- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

Whenever you change a CIF source file, regenerate a supervisor, change a hardware mapping, etc, you need to regenerate the PLC code as well. Follow these steps:

- Make sure the Config Mode is enabled, by switching to it if it is not the current mode.
- Save the entire solution in TwinCAT by clicking **File › Save All**.
- Regenerate PLC code from Eclipse. If using a script, wait for the execution of the script to finish (the status indicated as part of the title of the console should change from `running` to `FINISHED`).
- Switch back to TwinCAT (Microsoft Visual Studio).
- For all *File has been changed outside the environment. Reload the new file?* questions, click **[Yes]** or press `Enter`. See the Frequently Asked Questions (FAQ) for instructions on how to disable some of these questions.

- When a *File Modification Detected* window pops up for the project, click **[Reload]** to reload the updated project that was written by the CIF tooling.
- Clean the project by clicking **Build › Clean Solution**.
- Rebuild the project by clicking **Build › Rebuild Solution**.
- Make sure the hardware is powered up.
- Reload the I/O mappings:
  - In the project, under *I/O*, right click *Mappings*, and choose **Import Mapping Infos...**.
  - Select the mappings file for your hardware, and click **[Open]**.
  - If the import succeeds, you should see an *Untitled1 Instance - ...* item under the *Mappings* item.
- Activate the configuration:
  - Click **TwinCAT › Activate Configuration**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Activate Configuration (Old Configurations will be overwritten!)*. Click **[OK]** or press `Enter`.
  - A dialog pops up with the question *Restart TwinCAT System in Run Mode*. Click **[OK]** or press `Enter`.
- Start the PLC:
  - Click **PLC › Login**. Alternatively, click the appropriate toolbar item ().
  - A dialog pops up with the question *Application Port_851 does not exist on the device. Do you want to create it and proceed with download?*. Click **[Yes]** or press `Enter`.
  - Click **PLC › Start**. Alternatively, click the appropriate toolbar item ().
- The PLC is now running, and you can use the system.

Frequently Asked Questions (FAQ)

- **Question**: Is there something that can be done about the large number of *File has been changed outside the environment...* prompts?

  Answer: Yes. Click **Tools › Options...** to show the *Options* dialog. On the left, select **Environment › Documents**. Then, on the right, enable the *Detect when file is changed outside the environment* and *Auto-load changes, if saved* options.

- **Question**: Why doesn't importing the I/O mappings have any effect?

  Answer: Make sure the TwinCAT Standard PLC project is named `Untitled1`, and try again. Also, make sure you've built the solution before you try to import the I/O mappings.

- **Question**: I get a dialog with the warning *Device '...' needs a sync master (at least one variable linked to a task variable)*. What should I do?

Answer: You probably forgot to load an I/O mapping.

- **Question**: Why does opening the MAIN program take so long in Run Mode?

  Answer: As the generated code becomes larger, it can take a long time for TwinCAT to set up the environment for inline monitoring of the MAIN program. To disable inline monitoring, click **Tools › Options...** to show the *Options* dialog. On the left, select **TwinCAT › PLC Environment › Text Editor**. Then, on the right, switch to the *Monitoring* tab. Disable the *Enable inline monitoring option*.

- **Question**: TwinCAT hangs for long periods of time, before crashing and automatically being restarted by Windows. What should I do?

  Answer: If the generated code is too large, TwinCAT may not be able to handle it. In such cases, reduce the size of the generated code, by reducing the size of the CIF model for which you are generating the code.

- **Question**: I get an error stating *Reading file failed. [...]*. What should I do?

  Answer: The generated code is too large for TwinCAT to handle. See also the answer to the previous question.

S7 PLC output

The S7 output (1500, 1200, 400 and 300) generates code for SIMATIC controllers. Totally Integrated Automation Portal (TIA Portal) is an IDE for the development and testing of SIMATIC controllers. Using the S7 output, the generated PLC code can be imported in TIA Portal. This allows for tight integration and smooth development cycles when implementing controllers on SIMATIC controllers.

Generated output

The CIF PLC code generator can generate code files that can be imported in TIA Portal (see S7 usage).

The following option restrictions apply:

- The *Formal function invocation (arguments based)* option must be set to *For all functions* or *For functions with more than one argument*.
- The *Formal function invocation (function kind based)* option must be set to *For all functions*.
- The *Convert enumerations* option must be set to *Convert enumerations to integers* or *Convert enumerations to constants*.

The type of integer and real data types to use depends on the *PLC number bits* option. If *Automatic* is selected, the following data types are used:

S7 type	Integer type	Bits integer type	Real type	Bits real type
S7-1500	LInt	64	LReal	64
S7-1200	DInt	32	LReal	64
S7-400	DInt	32	Real	32
S7-300	DInt	32	Real	32

The *PLC task name*, *PLC task cycle time*, *PLC task priority*, *PLC project name*, *PLC configuration name*, and *PLC resource name* options have no effect for this output type.

S7 usage

Follow these steps to create a project in TIA Portal, import PLC code in it, and use the PLC code to control the hardware:

- Start TIA Portal.
- Create a new S7 project:
  - In the *Portal view*, click **Create new project**.
  - In the *Create new project* window, choose a project name and enter it in the *Project name* box.
  - For *Path*, select the directory in which the directory for the project should be created.
  - Click **[Create]** to create the project.
  - Switch to the *Project view*.
- Add the PLC device:
  - Unfold the project and select *Add new device*.
  - Select the device type (that is, S7-1500, S7-1200, S7-400 or S7-300).
  - Select the device with the correct article number.
  - Click **[OK]**.
- Import the generated PLC code (PLC tags):
  - Open *Default tag table* inside the *PLC tags* folder.
  - Click the *Import* button.
  - Select the *INPUTS.xml* file. Make sure *Tags* is selected and click **[OK]**.
  - Click the *Import* button, again.
  - Select the *CONSTS.xml* file. Make sure *Constants* is selected and click **[OK]**.
- Import the generated PLC code (Program blocks):
  - Unfold the *External source files* folder inside the device folder.
  - Click **Add new external files**.
  - Select all generated `.scl`, `.udt` and `.db` files (hold `Shift`) and click **[Open]**.

    - Select all imported files, right-click and select **Generate blocks from source**. Ignore any warnings or errors.
- Compile the project:
    - Right-click *Program blocks* and select **Compile › Software (rebuild all)**.
    - If there are warnings, compile again.
- Import timers (optional):
    - Sometimes, timers are not imported correctly. In that case, compilation fails. To add the timers manually, do the following.
    - Open *MAIN* inside the *Program blocks* folder.
    - Type `TON` and click `Enter`. The *Call options* window pops up.
    - In the *Call options* window, enter `timer0` in the *Name* box and click btn[OK].
    - Delete the generated lines of code.
    - Similarly, create `timer1`.
    - Delete the generated lines of code.
    - Compile the project again.
- Configure the inputs and outputs:
    - Open *Default tag table* inside the *PLC tags* folder.
    - On the *Tags* tab, enter the addresses for all the inputs. The syntax of the addresses is outside the scope of this manual.
    - On the *Tags* tab, create new entries for the output variables.
    - Open *MAIN* inside the *Program blocks* folder.
    - At the end of the program, after `END_WHILE`, assign values to the output variables. The values may originate from variables part of `"DB".state0`, variables and constants part of *Default tag table*, functions for algebraic variables and regular functions.
- Download PLC code to device:
    - Click the *Download to device* icon in the task bar. The *Extended download to device* window pops up.
    - Click **[Start search]**.
    - When a device has been found, click **[Load]**. The *Load preview* window pops up.
    - Click **[Load]**.
    - Change the *Start modules* action from *No action* to *Start module*.
    - Click **[Finish]**.

To debug the PLC code:

- Click *Go online* in the task bar.
- The values of various variables can now be inspected.
    - The values of input and output variables can be seen by opening *Default tag table* inside the

PLC tags folder. Click the *Monitor all* icon. The actual values appear in the *Monitor value* column.

- The values of state variables can be seen by opening *DB* under *Program blocks*. Click the *Monitor all* icon. Unfold the *state0* variable. The actual values appear in the *Monitor value* column.

To stop the PLC:

- While online, click **[STOP]** in the *CPU operation panel* in the *Testing* tab.
- Click *Go offline* in the task bar.

To reset the PLC and start from the initial state, with the same code and the same configuration:

- While online, click **[MRES]** in the *CPU operation panel* in the *Testing* tab. The *Online & diagnostics* window appears.
- Click **[Yes]**.
- If the PLC was in run mode, a new *Online & diagnostics* window appears. Click **[Yes]**.
- Click *Go online* in the task bar.
- Start the PLC by clicking **[RUN]** in the *CPU operation panel* in the *Testing* tab.
- The PLC is now running, and you can use the system.

4.6. Miscellaneous tools

4.6.1. CIF to CIF transformer

The CIF to CIF transformer can be used to syntactically transform CIF specifications to similar CIF specifications, which behave in the same manner (are semantically equivalent), but are expressed using different language constructs. For example, the CIF to CIF transformation that eliminates constants, replaces all uses of constants by their values, and removes the constants from the specification.

These reusable transformations are primarily used to increase the subset of CIF specifications that can be transformed or processed. Usually, there is no need to apply the CIF to CIF transformations manually. Transformations that work on subsets of the CIF language, automatically apply the necessary CIF to CIF transformations as a preprocessing step, to increase the translatable subset.

The CIF to CIF transformer can also be used as a pretty printer.

Starting the transformation

The transformation can be started from Eclipse in the following ways:

- Right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF miscellaneous tools › Apply CIF to CIF transformations...**.

- Right click an open text editor for a `.cif` or `.cifx` file and choose **CIF miscellaneous tools › Apply CIF to CIF transformations...**.
- Use the `cif2cif` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2cif` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output CIF specification. If not specified, the output file path defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.transformed.cif` file extension is added.
- *Transformations*: The CIF to CIF transformations to apply. The transformations are applied in the chosen order. Transformations may be applied multiple times. By default, no transformations are applied, essentially making the CIF to CIF transformer a pretty printer.

  In the option dialog, the available transformations are shown. To add a transformation, double click it, or select it and click the **[Add]** button below the list of available transformations. To remove a transformation, double click on it in the list of chosen transformations, or select it in that list and click the **[Remove]** button.

  At the command line, specify comma separated transformation names, in the order they should be applied. The names to use are listed below, in the Available transformations section.

Transformation descriptions

The pages describing the available transformations all use the same format. They have the following sections:

- *Introduction*: describes the transformation in a few sentences.
- *Supported specifications*: describes the subset of CIF specifications to which the transformation can be applied.
- *Preprocessing*: specifies the CIF to CIF transformations that are automatically applied as preprocessing, or the CIF to CIF transformations that can be manually applied as preprocessing, to increase the subset of CIF specifications that can be transformed by the transformation.
- *Implementation details*: further specifies what the transformation does, how it does it, and what the result of the transformation is. Most transformations also show one or more examples.
- *Renaming*: explains if and when renaming may be performed by the transformation.
- *Size considerations*: describes whether the transformation reduces or increases the size of the specification. If the specification may increase in size, an estimate of the order of increase is

given (linear, exponential, etc). Note that the size is not the file size, but rather the number of objects used to represent the specification (the number of automata, locations, edges, guards, binary operators, etc).

- *Optimality*: describes why the transformation result may not be optimal, if applicable. May also explain how to eliminate certain concepts that may be introduced as the result of the transformation.
- *Annotations*: describes which specific annotations the transformation processes, as well as whether the transformation adds or removes any annotations.

Available transformations

A whole collection of CIF to CIF transformations is available. The available transformations are listed below, with a link to the information about the transformation, and the name to use for that transformation on the command line.

- Add default initial values (`add-default-init-values`)
- Anonymize names (`anonymize-names`)
- Convert to interface (complete) (`convert-to-complete-interface`)
- Convert to interface (reduced) (`convert-to-reduced-interface`)
- Convert uncontrollable events to controllable (`convert-uncntrl-events-to-cntrl`)
- Convert controllable events to uncontrollable (`convert-cntrl-events-to-uncntrl`)
- Eliminate algebraic variables (`elim-alg-vars`)
- Eliminate automata to string casts (`elim-aut-casts`)
- Eliminate channels (`elim-channels`)
- Eliminate component definition/instantiation (`elim-comp-def-inst`)
- Eliminate constants (`elim-consts`)
- Eliminate equations (`elim-equations`)
- Eliminate groups (`elim-groups`)
- Eliminate `if` updates (`elim-if-updates`)
- Eliminate the use of locations in expressions (`elim-locs-in-exprs`)
- Eliminate monitors (`elim-monitors`)
- Eliminate automaton `self` references (`elim-self`)
- Eliminate state/event exclusion invariants (`elim-state-event-excl-invs`)
- Eliminate state invariants (`elim-state-invs`)
- Eliminate state invariants (plants only) (`elim-plant-state-invs`)
- Eliminate state invariants (supervisors only) (`elim-supervisor-state-invs`)
- Eliminate `tau` event (`elim-tau-event`)

- Eliminate tuple field projections (`elim-tuple-field-projs`)
- Eliminate type declarations (`elim-type-decls`)
- Convert enumerations to constants (`enums-to-consts`)
- Convert enumerations to integers (`enums-to-ints`)
- Lift events (`lift-events`)
- Linearize (merge) (`linearize-merge`)
- Linearize (product) (`linearize-product`)
- Linearize (product with merge) (`linearize-product-with-merge`)
- Merge enumerations (`merge-enums`)
- Push print file declarations into print declarations (`print-file-into-decls`)
- Relabel supervisors as plants (`relabel-supervisors-as-plants`)
- Remove annotations (`remove-annos`)
- Remove CIF/SVG declarations (`remove-cif-svg-decls`)
- Remove I/O declarations (`remove-io-decls`)
- Remove print declarations (`remove-print-decls`)
- Remove position information (`remove-pos-info`)
- Remove requirements (`remove-reqs`)
- Remove unused algebraic variables (`remove-unused-alg-vars`)
- Remove unused events (`remove-unused-events`)
- Simplify others (`simplify-others`)
- Simplify values (`simplify-values`)
- Simplify values (optimized) (`simplify-values-optimized`)
- Simplify values (no references) (`simplify-values-no-refs`)
- Simplify values (no references, optimized) (`simplify-values-no-refs-optimized`)
- Push SVG file declarations into other CIF/SVG declarations (`svg-file-into-decls`)
- Convert `switch` expressions to `if` expressions (`switches-to-ifs`)

Pretty printer

If no transformations are applied by the CIF to CIF transformer, the input CIF specification is just read from the input file, validated, and written to the output file. This can be useful, as it essentially turns the CIF to CIF transformer into a CIF *pretty printer*.

As a side effect of reading the input file and validating it, the model may change slightly. In fact, this occurs not only for the CIF to CIF transformer but for all tools, as they all read input files and validate them, before continuing with their actual task. Examples of such changes include the elimination of imports, the loss of comments, and the evaluation of tuple projection indices.

Add default initial values

This CIF to CIF transformation adds the default initial values to variables that are not given an explicit initial value. This includes discrete and continuous variables, as well as local variables of functions.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

The default values for all types are simple values. The only exception are function types, for which a new function is introduced. This new functions returns the default value for the return type of the function. If multiple default values for the same function type are needed, the newly introduced function is reused. The functions are named `_f`, `_f2`, `_f3`, etc.

For instance, the following specification:

```
automaton x:
  disc int x, y = 5;
  disc func bool (int) f;
  location;
end
```

is transformed to the following specification:

```
func bool _f(int p0):
  return false;
end

automaton x:
  disc int x = 0, y = 5;
  disc func bool (int) f = _f;
```

```
  location;
end
```

Renaming

If the names of the functions that are introduced, conflict with already existing declarations, they are renamed. For instance, if `_f2` is already in use, `_f22`, `_f23`, etc, are used instead. If renaming takes place, a warning is printed to the console.

Size considerations

Since values are added, this transformation increases the size of the specification. For an array of 400 elements, with an array of 300 integers each, the default initial value consists of 120,000 integers. Adding such an initial value can significantly increase the size of the model.

Optimality

The default values for all types are as primitive as they can be, and can usually not be simplified any further. It may however be possible that functions are introduced for the default values of function types, while such a function already existed in the original specification.

Annotations

This transformation does not process, add, or remove any annotations.

Anonymize names

This CIF to CIF transformation anonymizes the names of all named objects in the CIF specification.

Anonymization can be useful when the model is to be shared publicly, but contains confidential information. For instance, this can be useful when adding a confidential model to a bug report, when contributing it to model repositories, or when it is a part of a publication or thesis.

Only the names of named objects are anonymized. The resulting specification may still contain string literals, references to file names, etc, that are not anonymized by this transformation. It remains your own responsibility to decide whether or anonymizing names is sufficient to allow a model to be made public or not.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Automaton to string casts are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation.

Implementation details

The new names of objects are based on a fixed prefix that depends on what kind of object it is, followed by a number. The following table indicates the prefixes that are used for the different kinds of objects:

Prefix	Kind of object
`alg`	Algebraic variable
`aparam`	Algebraic parameter of a component definition
`aut`	Automaton
`autdef`	Automaton definition
`const`	Constant
`cparam`	Component parameter of a component definition
`disc`	Discrete variable
`enum`	Enumeration
`eparam`	Event parameter of a component definition
`evt`	Event
`field`	Field of a tuple type
`fparam`	Parameter of a user-defined function
`func`	User-defined function
`fvar`	Variable in an internal user-defined function
`grp`	Group
`grpdef`	Group definition
`input`	Input variable
`inv`	State (exclusion) invariants and state/event exclusion invariants
`inst`	Component instantiation
`iparam`	Input parameter of a component definition
`lit`	Enumeration literal
`loc`	Location of an automaton
`local`	Expression-local variables
`lparam`	Location parameter of a component definition

Prefix	Kind of object
`type`	Type declaration

For most kinds of objects, each unique object of that kind gets a unique number as postfix. For instance, consider a specification with algebraic variables and constants:

```
group g:
  alg bool at = true;
  alg bool af = false;
  const bool ct = true;
  const bool cf = false;
end

alg bool at = true;
alg bool af = false;
const bool ct = true;
const bool cf = false;
```

The result of anonymizing the names would be:

```
group grp1:
  alg bool alg1 = true;
  alg bool alg2 = false;
  const bool const1 = true;
  const bool const2 = false;
end

alg bool alg3 = true;
alg bool alg4 = false;
const bool const3 = true;
const bool const4 = false;
```

The four algrebraic variables each get a different number as postfix, as do the constants. This makes it easy to find a specific named object based on its name. But algebraic variables and constants are numbered independently.

For controllable and uncontrollable events, the new name is additionally prefixed with `c_` and `u_`, respectively. Events that are not controllable or uncontrollable don't get an additional prefix.

Compatible enumerations have the same number of literals, and the same literal names in the same order. The new literal names for compatible enumerations will be identical, to ensure they remain compatible.

To ensure that compatible tuple types remain compatible, their field names are not numbered globally, but are instead based on their index within the tuple type. That is, for a tuple type `tuple(int nr; real weight)` the anonymized result is `tuple(int field1; real field2)`.

Objects without a name, such as nameless locations or nameless invariants, are not given a name.

Renaming

Renaming is performed as explained in the previous section.

Size considerations

The only change to the specification is that objects are renamed. The new names could be shorter or longer than the original names.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. The names of annotations and annotation arguments are not anonymized. If the specification has a controller properties annotation, it is removed.

Convert to interface

This CIF to CIF transformation converts a specification to its *interface* (also known as *skeleton*). The interface contains the minimal elements to allow another specification to refer to its constants, types, events, functions and state. The generated interface can for instance be imported in other specifications, if it is undesired to import the full original specification.

The interface generated by the transformation has the same component structure as the original specification, but elements that cannot be referred to by name are removed. Some of the elements that are kept are replaced by simpler elements, such as algebraic variables that are replaced by input variables. There are two variants of this CIF to CIF transformation, one that generates the complete interface, and one that generates a reduced interface (without the functions and input variables of the original specification).

Example

To give an impression of how the transformation can be used, consider the following example `plant.cif` file, a plant model used for synthesis:

```
alg int[0..10] a = 8;
input bool i;
```

It defines an integer algebraic variable `a` and a boolean input variable `i`. The value of the algebraic

variable is fully defined in the model, while the value of the input variable is not defined in the model, but instead comes from the environment. The model is therefore an *open* model, with input variable `i` defining the *interface* to the environment.

If we implement this plant (or the supervisor synthesized from it), we can connect the input variable to the environment of the system. For instance, if we generate PLC code, we can connect it to a PLC input. Similarly, if we simulate the model, we may assign values to the input variable through clicks on an SVG image. Either the PLC or the SVG image then provides a concrete environment for the model, implementing the interface. The environment determines the value of the variable during execution, thus *closing* the model.

However, in some cases, we may not want to get the value(s) of an input variable from the environment. This can for instance be useful when the plant model is to be simulated, but the input is not provided by the user. Instead, a different (part of the) system — that is outside the scope of the plant model — provides values.

In such cases, the simulation model can be extended, such that it plays the role of the environment. This approach also closes the plant (or supervisor) model.

We may for instance make a simulation model that contains:

```
alg bool i = true;
```

This model gives a concrete value to `i`, by defining it as an algebraic variable. To simulate the plant without user interaction, we need to combine the plant model (or the synthesized supervisor) with the simulation model into a single closed model. We can use the CIF merger tool for this. Note that the separate plant and simulation models are preserved, and we can thus still perform synthesis on the plant model.

In our example case, the CIF merger merges the input variable of the plant model with the algebraic variable of the simulation model:

```
alg int[0..10] a = 8;
alg bool i = true;
```

All input variables have then been made concrete, giving us the single closed model that we can simulate (in this case completely without user interaction).

We would prefer to have a single model of the plant for synthesis and simulation, rather than two separate models, but that is not possible:

- For synthesis, we want to explicitly leave the value of `i` to be decided by the environment, and we therefore model `i` as input variable in the plant model.
- For simulation, we want to concretely define the value of `i` in the model itself, and we therefore model `i` as an algebraic variable in the simulation model.
- Variable `i` can't be both an input variable and an algebraic variable in the same model. Therefore, a single model is not possible.

Another reason for having to use two separate models, is that for simulation we may want to use features to specify the value(s) of `i` that are not supported by synthesis. We could for instance want to use a continuous variable in the simulation model:

```
cont c = 0 der 1;
alg bool i = c >= 10 and c <= 20;
```

The situation becomes more complex when the simulation model is also open. In such a case, it also needs values from its environment, like in this simulation model:

```
cont c = 0 der 1;
alg bool i = a > 3 and c >= 10 and c <= 20;
```

Since `i` is now defined in terms of both `c` and `a`, we need `a` to be available in the simulation model as well, as without it the model is invalid. Since `a` exists in the plant model, intuitively the solution seems to be to have the simulation model import the plant (or supervisor) model it aims to extend:

```
import "plant.cif";
cont c = 0 der 1;
alg bool i = a > 3 and c >= 10 and c <= 20;
```

However, `i` is then defined twice: once as algebraic variable (from the simulation model) and once as input variable (from the plant model). As this is not allowed, the model is invalid. Instead of importing the plant model into the simulation model, we can resolve the missing `a` in the simulation model by defining it as an input variable:

```
input int[0..10] a;
cont c = 0 der 1;
alg bool i = a > 3 and c >= 10 and c <= 20;
```

This is a nice solution. The plant model now concretely defines `a` and has `i` as an interface, while the simulation model concretely defines `i` and has `a` as an interface. Each variable is only defined concrete values in one model, and the part of the other model that it needs — but does not want to define — is specified as an interface, providing the minimal information of what it needs. In this way, changes to the concrete values have to made only in one model.

Both the plant model and the simulation model are open models. To simulate them, we can still use the CIF merger to merge them into a single closed model. For the example, merging the models results in:

```
alg int[0..10] a = 8;
cont c = 0 der 1;
alg bool i = a > 3 and c >= 10 and c <= 20;
```

While both the plant and simulation models now look good and work as desired, the simulation model now contains the interface of the plant. Input variable `a` in the simulation model duplicates the relevant information about algebraic variable `a` from the plant model. Such duplicates must be kept consistent between the plant model and the simulation model. For instance, if algebraic variable `a` is renamed in the plant model, we have to rename input variable `a` in the simulation model. If we forget to do so, then the simulation model is still valid, but merging the plant and simulation models does not merge the variables anymore, and the simulation will no longer work as expected. Manually updating the plant interface in the simulation model can become tedious when the plant model grows or when large changes occur.

One of the problems in updating the plant interface in the simulation model, is that the simulation model contains both elements for simulation and elements that are part of the duplicated plant interface. We can split off the plant interface part to a separate file, say `plant_interface.cif`, and import that new file in the simulation model. Then it becomes easier to compare the plant model against its interface, and to keep them aligned.

However, the `plant_interface.cif` file still needs to be manually adapted if the plant changes. This *Convert to interface* CIF to CIF transformation solves that problem, as it automatically generates the plant interface from the plant model. For our example, we use the variant that generates a reduced interface. The reduced interface contains only the part of the interface of the plant that is concretely defined in the plant model, avoiding a duplicate definition of `i`. The transformation generates the following reduced interface:

```
input int[0..10] a;
```

This reduced interface model can be imported into the simulation model, as follows:

```
import "plant_interface.cif";
cont c = 0 der 1;
alg bool i = a > 3 and c >= 10 and c <= 20;
```

This simulation model still defines `i` concretely, and has access to `a` defined in the plant model. Since we have not modified the plant model, we can still use it for synthesis. We can also still merge the plant model (or synthesized supervisor) with the simulation model as before, and the merged model is also still as it was before. Furthermore, the plant interface is no longer manually duplicated in the simulation model, reducing the effort to keep the simulation model consistent with the plant model as the plant model evolves.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

No preprocessing is performed by this CIF to CIF transformation.

Implementation details

The following are removed from the specification:

- Annotations.
- I/O declarations.
- Initialization and marker predicates (in components).
- Invariants (in components).
- Equations (in components).
- Functions (only if a reduced interface is generated).
- Input variables (only if a reduced interface is generated).
- Locations without a name.
- All parameters of component definitions, except for component parameters.
- All arguments of component instantiations for which the corresponding parameters of the instantiated component definitions are removed.

I/O declarations are removed as you can't refer to them. Similarly, initialization and marker predicates, invariants and equations are removed. And also locations without a name are removed.

The parameters of component definitions are removed, as you can't refer to them from outside the definitions. The corresponding arguments of component instantiations are then also removed, to ensure the model remains valid. Component parameters of component definitions are preserved, as there could be references to types via these component parameters. Since the component structure is preserved, these can be kept in the interface models.

Functions are only removed if the variant of the transformation is used that generates a reduced interface. If the variant is used that generates a complete interface, then they are obviously not removed.

Input variables are only removed if the variant of the transformation is used that generates a reduced interface. If the variant is used that generates a complete interface, then they are obviously not removed. Note that input variables generated by the transformation (see below) are never removed, regardless of which variant is used, as only input variables from the input model are removed.

Some elements of the specification are replaced:

- An algebraic variable is replaced by an input variable that has the same name and type.
- A continuous variable is replaced by an input variable that has the same name and a `real` type.
- A discrete variable is replaced by an input variable that has the same name and type.

- A location of an automaton with a name is replaced by an input variable that has the same name and a `bool` type.
- An automaton is replaced by a group with the same name.

Algebraic, continuous and discrete variables are replaced by input variables. The values of these variables are thus not specified in the interface specification. Still, this is sufficient to be able to refer to them, and thus serves as a proper interface.

Similarly, locations are replaced by input variables. Everything within locations, such as initialization predicates, marker predicates and edges, are thus removed. The contents of the location can't be referred to, and the input variable is sufficient as interface for the location.

Automata are replaced by groups, to preserve the structure of the model, and to ensure that the declarations within the automata, such as their locations that become input variables, keep their absolute names. The declarations of the automata, such as types, constants, and discrete variables turned input variables, are all preserved in the groups. The other details of the automata, such as their alphabets and monitors, are removed. These details are also not relevant for an interface model, as you can't refer to them. Similar to automata becoming groups, also automaton definitions become group definitions.

Note that various elements of the specification are explicitly kept:

- Component definitions and instantiations.
- Constants.
- Enumerations.
- Events.
- Groups.
- Input variables (only if a complete interface is generated).
- Type declarations.
- User-defined functions (only if a complete interface is generated).

The resulting interface specification may consist of these elements as well:

- Annotations (only a single `@@partial` annotation, see below).
- Component definitions and instantiations.
- Constants.
- Enumerations.
- Events.
- Groups.
- Input variables.
- Type declarations.
- User-defined functions (only if a complete interface is generated).

If a reduced interface of a model is imported into another model, that other model can be merged with the original model, assuming nothing else in the other model prevents such a merge.

Renaming

n/a

Size considerations

Various elements of the specification are removed, or replaced by simpler elements, reducing the size of the specification.

Optimality

n/a

Annotations

This transformation removes all annotations from the specification. Subsequently, a partial specification annotation is added to the specification.

Convert uncontrollable events to controllable

This CIF to CIF transformation converts uncontrollable events to controllable.

As `tau` is neither controllable nor uncontrollable, it is not touched by this transformation.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

Each uncontrollable event in the CIF specification is converted to controllable. For instance, the following specification:

```
uncontrollable a;
controllable b;

automaton aut:
  location l1:
    initial;
    marked;
    edge a goto l2;
  location l2:
    edge b goto l1;
end
```

is transformed to the following specification:

```
controllable a;
controllable b;

automaton aut:
  location l1:
    initial;
    marked;
    edge a goto l2;
  location l2:
    edge b goto l1;
end
```

Renaming

Event names with the special prefix `u_` are renamed to preserve the event naming convention, see Event controllability. The prefix `u_` is replaced by `c_`. In case the new global name is already in use in the specification, a suffix is also added.

Size considerations

Uncontrollable events are modified, the size of the specification neither grows nor shrinks.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. If the specification has a controller properties annotation, it is removed.

Convert controllable events to uncontrollable

This CIF to CIF transformation converts controllable events to uncontrollable.

As `tau` is neither controllable nor uncontrollable, it is not touched by this transformation.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

Each controllable event in the CIF specification is converted to uncontrollable. For instance, the following specification:

```
uncontrollable a;
controllable b;

automaton aut:
  location l1:
    initial;
    marked;
    edge a goto l2;
  location l2:
    edge b goto l1;
end
```

is transformed to the following specification:

```
uncontrollable a;
uncontrollable b;

automaton aut:
  location l1:
    initial;
    marked;
    edge a goto l2;
  location l2:
```

```
    edge b goto l1;
end
```

Renaming

Event names with the special prefix `c_` are renamed to preserve the event naming convention, see Event controllability. The prefix `c_` is replaced by `u_`. In case the new global name is already in use in the specification, a suffix is also added.

Size considerations

Controllable events are modified, the size of the specification neither grows nor shrinks.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. If the specification has a controller properties annotation, it is removed.

Convert `switch` expressions to `if` expressions

This CIF to CIF transformation converts `switch` expressions to `if` expressions.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

All `switch` expressions are replaced by `if` expressions. For instance, consider the following CIF specification:

```
alg int y = ...
alg real x = switch y:
               case 1: 1.0
               case 2: 4.0
               case 3: 9.0
               else    0.0
             end;
alg real z = switch y:
               else 1.0
             end;
```

```
automaton a:
  alg string v = switch self:
                   case l1: "first"
                   case l2: "second"
                 end;

  location l1:
    initial;
    edge tau goto l2;

  location l2;
end
```

This is transformed to the following CIF specification:

```
alg int y = ...
alg real x = if   y = 1: 1.0
             elif y = 2: 4.0
             elif y = 3: 9.0
             else        0.0
             end;
alg real z = 1.0;

automaton a:
  alg string v = if l1: "first"
                 else   "second"
                 end;

  location l1:
    initial;
    edge tau goto l2;

  location l2;
end
```

For algebraic variable `z`, there is only one case (the `else`), so no `if` expression is generated. This optimization means that the control value (`y` in this case) is lost.

For algebraic variable `v` in automaton `a`, the `switch` expression uses an automaton self reference. The cases list the possible locations of the automaton, and the `switch` expression maps them to string values. Since for `if` expressions the `else` part is mandatory, the last `case` of the `switch` is converted to the `else` of the `if` expression.

Renaming

n/a

Size considerations

The created `if` expression is usually larger than the original `switch` expression, as the control value is duplicated for each `if` and `elif` part, where it is compared to the `case` values. For `switch` expressions with only one `case` or `else`, no `if` expression is generated, and the expression becomes smaller.

Optimality

This transformation transforms `switch` expressions to `if` expressions, leading to `if` expressions with `if` and `elif` guards that may not be optimal. To simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate algebraic variables

This CIF to CIF transformation eliminates algebraic variables and their equations.

See also: Eliminate equations.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Automaton `self` references are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation
- Eliminate automaton `self` references

Implementation details

For each algebraic variable with a single equation in a component or automaton, all uses of the variable (for instance in guards, invariants, etc) are replaced by the value of the variable (the right hand side of its equation). For algebraic variables with equations specified in the locations of an

automaton, an 'if' expression is created. This 'if' expression uses the locations of the automaton as guards, and the right hand sides of the equations as values for each of the alternatives. The last alternative is an 'else', and thus has no guard. If the automaton has only one location, generation of the 'if' expression is suppressed.

The algebraic variables and their equations are removed.

For instance, the following specification:

```
alg bool b1 = true;
alg bool b2;
equation b2 = false;

automaton aut:
  alg int a;
  location l1:
    equation a = 1;
  location l2:
    equation a = 2;
end

invariant b1 or b2 or p.a > 0;
```

is transformed to the following specification:

```
automaton aut:
  location l1;
  location l2;
end

invariant true or false or if p.l1: 1 else 2 end > 0;
```

Renaming

n/a

Size considerations

Since algebraic variables are shortcuts for expressions, eliminating them could result in an increase of the size of the specification. Algebraic variables may be defined in terms of other algebraic variables. Therefore, in the worst case, the increase is exponential.

Optimality

For an assignment `x := 1 + y`, where `y` is an algebraic variable that is eliminated, and where `y` has

value `5`, the resulting assignment is `x := 1 + 5`. The result is not simplified any further. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

This transformation introduces 'if' expressions for algebraic variables that have an equation per location, for automata with at least two locations. The guards of such 'if' expressions refer to the locations of the automata. To eliminate such location references, apply the Eliminate the use of locations in expressions CIF to CIF transformation.

Annotations

This transformation does not process or add any annotations. As this transformation removes all algebraic variables from the specification, their annotations are removed as well.

Eliminate automata to string casts

This CIF to CIF transformation eliminates casts of automata to strings.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All casts of automata to strings, using either implicit automaton `self` references or explicit automata references, are eliminated. If the referred automaton has only a single location, the cast is replaced by a string literal with the name of that location. If the automaton has multiple locations, the cast is replaced by an `if` expression, using the locations of the automaton as guards, and string literals containing the location names as values. For nameless locations the string literal `"*"` is used.

For instance, consider the following CIF specification:

```
automaton a:
  alg string x = <string>self;

  location:
```

```
    initial;
end

automaton b:
  alg string x = <string>self;

  location l1:
    initial;
  location l2:
end
```

This is transformed to the following CIF specification:

```
automaton a:
  alg string x = "*";

  location:
    initial;
end

automaton b:
  alg string x = if   l1: "l1"
                 elif l2: "l2"
                 else     "l3"
                 end;

  location l1:
    initial;
  location l2;
  location l3;
end
```

Renaming

n/a

Size considerations

The created `if` expressions are usually larger than the original cast expressions. The increase is linear in terms of the number of locations times the number of casts.

Optimality

This transformation introduces 'if' expressions for automaton to string casts, for automata with at least two locations. The guards of such 'if' expressions refer to the locations of the automata. To eliminate such location references, apply the Eliminate the use of locations in expressions CIF to

CIF transformation.

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate channels

This CIF to CIF transformation eliminates channels, introducing a regular/non-channel event per send/receive combination.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Specifications with SVG input mappings that may map to a channel are currently not supported.

Preprocessing

To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations if applicable:

- Eliminate component definition/instantiation
- Remove I/O declarations

Implementation details

Each channel is transformed into as many events as there are pairs of senders and receivers for the event. So, in case a channel has *m* send options and *n* receive options, *m* * *n* new events are created to replace it.

For instance, consider the following CIF specification:

```
event int chan;

automaton sender1:
  location first:
    initial;
    edge chan!1 goto second;

  location second:
    edge chan!2 goto first;
end

automaton sender2:
  location first:
    initial;
    edge chan!3 goto second;
```

```
  location second:
    edge chan!4 goto first;
end

automaton receiver1:
  location:
    initial;
    edge chan?;
end

automaton receiver2:
  location:
    initial;
    edge chan?;
end

automaton synchronizer:
  location:
    initial;
    edge chan;
end
```

This transformation transforms it into a new specification equivalent to the following:

```
event chan_1;
event chan_2;
event chan_3;
event chan_4;
event chan_5;
event chan_6;
event chan_7;
event chan_8;

automaton sender1:
  location first:
    initial;
    edge chan_1, chan_2 goto second;

  location second:
    edge chan_3, chan_4 goto first;
end

automaton sender2:
  location first:
    initial;
    edge chan_5, chan_6 goto second;

  location second:
    edge chan_7, chan_8 goto first;
end
```

```
automaton receiver1:
  location:
    initial;
    edge chan_1;
    edge chan_3;
    edge chan_5;
    edge chan_7;
end

automaton receiver2:
  location:
    initial;
    edge chan_2;
    edge chan_4;
    edge chan_6;
    edge chan_8;
end

automaton synchronizer:
  location:
    initial;
    edge chan_1, chan_2, chan_3, chan_4,
         chan_5, chan_6, chan_7, chan_8;
end
```

There are two send automata for channel `chan`, each with two send options, making for four send options. There are two receive automata, each with one receive option, making for two receive options. There are thus 4 * 2 = 8 send/receive combinations, and then 8 replacement events are created.

In the synchronizing automaton, you see the channel reference on the edge is replaced by all 8 replacement events. In the sender and receiver automata, you see the channel references are replaced only by the relevant replacement events for which the send or receive option participates.

In general, if state/event exclusion invariants, explicit alphabets of automata, monitors of automata, print declaration `for` declarations, and edge events on edges of automata, refer to the channel, then these references are replaced by references to the new replacement events. For a state/event exclusion invariant that applies to a channel, the invariant is cloned as many times as there are replacement events for the channel.

If a channel has no senders or no receivers, it is removed entirely, including all references to it. All state/event exclusion invariants for the channel are then removed entirely. If an explicit monitor declaration of an automaton becomes empty due to removing event references from it, the monitor declaration is removed to prevent the automaton from monitoring all events in its alphabet. If an implicit monitor declaration no longer has an effect when an automaton's alphabet becomes empty, the monitor declaration is removed to prevent warnings about monitoring an empty alphabet. Similarly, if the `for` declarations of a print declaration become empty, the print declaration is removed to prevent the default `for` declarations from taking effect. And if all events on an edge are

removed, the edge is removed to prevent it from becoming a `tau` edge.

For edges with a receive, all occurrences of the received value expression (`?`) in updates, are replaced by the send value expression of the relevant send option of the corresponding send/receive combination. Edges with multiple receives on them are first split, so that each edge has a single receive on it. For each edge with a receive, the edge is furthermore duplicated for each send/receive edge event combination that applies to it.

Renaming

This transformation creates new events to replace channels. If the name of a new event is already in use, the new event is given a different name to prevent conflicts. In case of such conflicts, a warning is printed to the console.

Size considerations

In case a channel has *m* send options and *n* receive options, *m* * *n* new events are created to replace it. This may result in a blow-up of the size of the specification, as there may be a quadratic number of events for each channel, given the number of send and receive options they have.

In places where the channel was previously referred, now multiple events may need to be referred.

In some cases, elements of the specification may be duplicated, once per replacement event. For instance, a state/event exclusion with a channel event is duplicated *m* * *n* times.

If for a channel there are no send options or no receive options, elements of the specification may get removed, reducing the size of the specification.

Optimality

If a channel is superflously referred multiple times in for instance the alphabet or monitor specification of an automaton, all references to the channel are replaced once by the replacement events, removing the redundancy.

Annotations

This transformation does not process or add any specific annotations. The annotations of channels are removed. If any other elements of the specification get removed, their annotations are also removed.

Eliminate component definition/instantiation

This CIF to CIF transformation eliminates all component definitions and instantiations, by actually instantiating the component definitions.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

This transformation operates in three phases:

- Phase 1: Find the component definitions without any component definitions and/or component instantiations in them.
- Phase 2: Instantiate the component definitions found in phase 1, by putting the body of the definition in the place of the instantiation, in a component named after the component instantiation. Also removes the component definitions that were just instantiated, and introduces new local algebraic variables for the algebraic parameters.
- Phase 3: Replace uses of component, event, input, and location parameters by the arguments provided for the instantiation.

For instance, the following specification:

```
group def P(alg int x):
  invariant x > 0;
end

p1: P(1);

alg int y = 2;
p2: P(y);
```

is transformed to the following specification:

```
group p1:
  alg int x = 1;
  invariant x > 0;
end

alg int y = 2;
group p2:
  alg int x = y;
  invariant x > 0;
end
```

Absolute references may be needed to express the results of this transformation. For instance:

```
const int x = 5;

group def X():
  invariant x = 5;
end
```

```
group a:
  x: X();
end
```

is transformed to the following specification:

```
const int x = 5;

group a:
  group x:
    invariant .x = 5;
  end
end
```

Note how the invariant in component `a.x` can not refer to constant `x` directly, as `x` refers to component `a.x` in that context. Therefore, the scope absolute reference `.x` is used instead.

Renaming

n/a

Size considerations

Since component definitions are shortcuts for components, eliminating them could result in an increase of the size of the specification.

Optimality

n/a

Annotations

This transformation does not add any new annotations. Each instantiated component gets the annotations of its component definition and instantiation. If a component definition is instantiated multiple times, this thus leads to a duplication of the annotations of that definition on its instantiated components. The annotations of automaton definitions that are not instantiated are removed. As each algebraic parameter becomes an algebraic variable when a component definition is instantiated, the variable gets the annotations of the parameter.

Eliminate constants

This CIF to CIF transformation eliminates constants.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All uses of constants (for instance in guards, invariants, etc), are replaced by their values. The constants themselves are removed. For instance, the following specification:

```
const int x = 1;
const int z = y;
const int y = x + 1;
invariant x + y + z > 0;
```

is transformed to the following specification:

```
invariant 1 + 2 + 2 > 0;
```

Renaming

n/a

Size considerations

Since constants are shortcuts for values, eliminating them could result in an increase of the size of the specification. Constants may be defined in terms of other constants. Therefore, in the worst case, the increase is exponential.

Optimality

For an assignment `x := 1 + y`, where `y` is a constant that is eliminated, and where `y` has value `5`, the resulting assignment is `x := 1 + 5`. The result is not simplified any further. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Annotations

This transformation does not process or add any annotations. As this transformation removes all constants from the specification, their annotations are removed as well.

Eliminate equations

This CIF to CIF transformation eliminates equations for algebraic variables and derivatives of continuous variables. Note that it does not eliminate the variables themselves.

See also: Eliminate algebraic variables.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Automaton `self` references are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation
- Eliminate automaton `self` references

Implementation details

For algebraic variables with a single equation in a component or automaton, the value of the variable (the right hand side of the equation) is moved to the declaration of the algebraic variable. For algebraic variables with equations specified in the locations of an automaton, an 'if' expression is created. This 'if' expression uses the locations of the automaton as guards, and the right hand sides of the equations as values for each of the alternatives. The last alternative is an 'else', and thus has no guard. The single right hand side, or the 'if' expression, is put directly in the declaration of the algebraic variable. If the automaton has only one location, generation of the 'if' expression is suppressed.

The derivatives of continuous variables are transformed in a similar way. The derivative ends up in the declaration of the continuous variable.

All equations are removed from components (including automata) and locations.

For instance, the following specification:

```
alg bool b1 = true;
cont x;
equation x' = 1.0;

automaton p:
  alg int a;
  location l1:
    equation a = 1;
  location l2:
    equation a = 2;
end
```

is transformed to the following specification:

```
alg bool b1 = true;
cont x der 1.0;

automaton p:
  alg int a = if l1: 1
              else   2
              end;
  location l1;
  location l2;
end
```

Renaming

n/a

Size considerations

The right hand sides of the equations are simply moved. This does not influence the size of the specification. For equations specified in the locations of an automaton, an 'if' expression is created. This increases the size of the specification. The increase however, is linear in the number of equations.

Optimality

This transformation introduces 'if' expressions for algebraic variables and derivatives of continuous variables that have an equation per location of an automaton, for automata with at least two locations. The guards of such 'if' expressions refer to the locations of the automata. To eliminate such location references, apply the Eliminate the use of locations in expressions CIF to CIF transformation.

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate groups

This CIF to CIF transformation flattens the component structure of a specification, by eliminating all groups. That is, after this transformation, all automata will be at the top level, directly in the specification.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Furthermore, this transformation does not guarantee the correct result for specification with enumerations. See also the Renaming section below.

Preprocessing

To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order) prior to using this transformation:

- Eliminate component definition/instantiation
- Convert enumerations to integers

The following CIF to CIF transformations are automatically applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Push SVG file declarations into other CIF/SVG declarations
- Push print file declarations into print declarations

Implementation details

All declarations, invariants, initialization predicates, marker predicates, and equations from the groups are lifted to the top level of the specification. The automata are lifted to the top level of the specification as well. Subsequently, the groups are removed.

In order to be able to track the origin of automata, declarations and invariants, they are named after their absolute names. For a group `x`, with a group `y`, with a constant `z`, the absolute name of the constant is `x.y.z`. The constant is lifted to the top level of the specification, and named `x_y_z`.

For instance, this specification:

```
group a:
  const int x = 5;
  invariant inv: x > 4;
end

group p:
  automaton q:
    event e;
    location;
  end
end
```

is transformed to the following specification:

```
const int a_x = 5;
invariant a_inv: a_x > 4;

automaton p_q:
  event e;
  location;
end
```

Renaming

Since declarations (and invariants) of the specification are merged with the declarations of the groups and with automata, renaming may be necessary to ensure uniquely named declarations. For instance, for a group `x`, with a group `y`, with a constant `z`, the absolute name of the constant is `x.y.z`. The constant is lifted to the top level of the specification, and renamed to `x_y_z`. However, if the specification already contains a declaration of another constant, automaton, etc named `x_y_z`, then the lifted constant is renamed to `x_y_z2` instead. If that name is also already in use, it is renamed to `x_y_z3`, etc.

Whenever renaming takes place, a warning will be printed to the console.

Renaming of enumeration literals

Enumeration literals that are part of lifted enumeration declarations, are not renamed to their absolute names. To understand why this is necessary, consider:

```
group x:
  enum E = A, B;
end

automaton y:
  enum E = A, B;
  location:
```

```
    initial;
end

invariant x.A = y.A;
```

Here, group `x` and automaton `y` both have an enumeration `E` with enumeration literals `A` and `B`. The invariant of the specification compares the literals of the two different enumerations. This is allowed, since the enumerations are *compatible* (they have the same enumeration literals, with the same names, in the same order). If we were to rename the enumeration literals to absolute names as we lift them to the top level of the specification, the result would be:

```
enum x_E = x_A, x_B;

automaton y:
  enum E = A, B;
  location:
    initial;
end

invariant x_A = y.A;
```

Which would no longer be a valid specification, as enumerations `x_E` and `y.E` are no longer compatible (they have enumeration literals with different names). Since we leave the names of the enumeration literals as they are, the result of the transformation is:

```
enum x_E = A, B;

automaton y:
  enum E = A, B;
  location:
    initial;
end

invariant A = y.A;
```

which is still a correct specification, and has the same meaning as the original specification.

Now assume the following specification:

```
group x:
  enum E = A, B;
end

group y:
  enum E = A, B;
end
```

```
invariant x.A = y.A;
```

Here, the two enumerations are both in groups. If we lift them, we have two enumerations, with the same literals, which is not a valid CIF specification. As such, renaming can not be avoided, and the result of the transformation will be:

```
enum x_E = A, B;
enum y_E = A2, B2;

invariant A = A2;
```

This result is an invalid CIF specification. Whenever an enumeration literal is renamed, a warning is printed to the console, informing of the possibility that the resulting specification is invalid. As such, it is highly recommended to avoid such situations, and eliminate the enumerations prior to the elimination of the groups.

Size considerations

The groups are removed. Other objects are lifted, which essentially means they are moved. The size of the specification does not increase.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. As this transformation removes all groups from the specification, their annotations are removed as well. If the specification has a controller properties annotation, it is removed.

Eliminate `if` updates

This CIF to CIF transformation eliminates 'if' updates on edges and in SVG input mappings.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Multi-assignments and partial variable assignments are not supported. That is, the left hand sides of the assignments on edges and in SVG input mappings (the addressables) must be single variables without projections.

Preprocessing

No preprocessing, to increase the subset of CIF specifications that can be transformed, is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

This transformation ensures that each variable that is assigned, gets a single explicit assignment. Simply put, the 'if' update structure is replicated by an 'if' expression. For instance, the following updates:

```
if g:
  if g2:
    x := 5
  else
    y := 6
  end
end,
z := 7
```

are transformed to the following updates:

```
x := if g:
       if g2:
         5
       else
         x
       end
     else
       x
     end,
y := if g:
       if g2:
         y
       else
         6
       end
     else
       y
     end,
z := 7
```

Observe how variables `x` and `y` are assigned inside two `if` updates. These `if` updates are replicated by two `if` expressions.

Since for `if` expressions the `else` part is not optional, dummy `x := x` assignments are sometimes performed in cases where the original specification would not have assigned variable `x` at all. That is, variables may be assigned their old values, resulting in superfluous assignments.

Renaming

n/a

Size considerations

The 'if' structure may be replicated for multiple variables, which may result in an increase of the specification size. The increase amount is influenced by the size of the guard expressions, and the number of levels of nesting 'if' updates.

Optimality

The superfluous assignments (see the Implementation details section above), are obviously not optimal. The language however requires them.

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate the use of locations in expressions

This CIF to CIF transformation eliminates the use of locations in expressions (such as guards, invariants, and equations), by introducing location pointer variables for automata, and using them instead.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

For each automaton with two or more locations, and for which a location is used (referred to) in an expression (such as a guard, invariant, or equation), a location pointer variable is introduced, with as value the current location of the automaton. The values for such a new variables are part of a newly created enumeration that has a value (an enumeration literal) for each location of the automaton. For automata with exactly one location, no location pointer variable is created.

To initialize the new location pointer variable, the variable itself is initialized to `any` (meaning any value in its domain), and initialization predicates are added to all locations that could potentially be initial locations. For automata with exactly one initial state, the initial value of the location pointer variable is set directly, instead of using initialization predicates.

All edges in the automaton that change the current location of that automaton, get an additional assignment to update the location pointer variable.

All uses of the locations of the automaton in expressions (that is, all location reference expressions) are changed to equality binary expressions for the location pointer variable and the enumeration literal corresponding to the location. For references to a location of an automaton with exactly one location, all uses of this location are changed to `true`.

For instance, the following specification:

```
automaton x:
  event e;
  alg bool a = l1;

  location l1:
    initial;
    edge e goto l2;

  location l2:
    edge e;
end
```

is transformed to the following specification:

```
automaton x:
  event e;
  alg bool a = LP = LOC_l1;
  disc LPE LP = LOC_l1;
  enum LPE = LOC_l1, LOC_l2;

  location l1:
    initial;
    edge e do LP := LOC_l2 goto l2;

  location l2:
    edge e;
```

```
end
```

Location pointer variable `LP` is added. It is initialized directly in the declaration, since there is exactly one initial location in this automaton. Enumeration `LPE` is created to represent the possible values of the location pointer variable. Enumerations literals `LOC_l1` and `LOC_l2` represent locations `l1` and `l2`, respectively.

The edge from location `l1` to location `l2` is extended with an update to the location pointer. The self loop in location `l2` is not extended, as the location does not change.

Renaming

If the names of the location pointer variables that are introduced, conflict with already existing declarations, they are renamed. For instance, if `LP` is already in use, `LP2`, `LP3`, etc, are used instead. Similarly, renaming is performed for enumerations, and enumeration literals that conflict with already existing declarations. If renaming takes place, a warning is printed to the console.

Size considerations

The number of added location pointer variables is linear in the number of automata. The number of added initialization predicates for the location pointer variables is linear in the number of possible initial locations. The number of added enumerations is linear in the number of automata. The number of added enumeration literals is linear in the number of locations. The number of added updates is linear in the number of edges. The number of added equality binary expressions is linear in the number of location references.

Optimality

This transformation introduces enumerations for the possible values of the location pointer variables. To eliminate the enumerations, apply the Convert enumerations to integers or Convert enumerations to constants CIF to CIF transformation.

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate monitors

This CIF to CIF transformation eliminates monitor events.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

For each monitor event in an automaton, additional self loops are added as needed, to make sure the automaton doesn't disable the monitor event. Furthermore, all monitors are removed from all automata.

The self loop edges created by this transformation have no communication (no `!...`, no `?...`), no urgency (no `now`), and no updates (no `do ...`).

For instance, for the following locations and edges, for monitor event `e`:

```
location l1:
  edge e goto l2;
  edge f when x = 2 goto l3;

location l2:
  edge e when x = 1 goto l3;
  edge e when x = 2 goto l1;

location l3;
```

the result after this transformation is:

```
location l1:
  edge e goto l2;
  edge f when x = 2 goto l3;

location l2:
  edge e when x = 1 goto l3;
  edge e when x = 2 goto l1;
  edge e when not(x = 1 or x = 2);

location l3:
  edge e;
```

In location `l1`, monitor event `e` is always enabled (no guard implies a `true` guard), and thus no additional self loop is needed. In location `l2`, event `e` is only enabled if `x` has either value `1` or value `2`. A self loop is added for all other cases, to ensure the event is always enabled. In location `l3`, the event is never enabled (no edges, which essentially means guard `false`). A self loop is added, to ensure the event is always enabled.

Event `f` is not a monitor event, and is thus not affected by this transformation.

Renaming

n/a

Size considerations

Since self loop edges may be added by this transformation, the size of the specification may increase.

Optimality

For an edge with guard `x = 1`, a self loop with guard `not (x = 1)` may be generated by this transformation. The guard could be simplified to `x != 1`. However, this transformation does not simplify guards. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Annotations

This transformation does not process, add, or remove any annotations. The newly added edges do have any annotations.

Eliminate automaton `self` references

This CIF to CIF transformation replaces implicit automaton `self` references by explicit automaton references.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given

order):

- Eliminate component definition/instantiation

Implementation details

All implicit automaton `self` references are replaced by explicit automaton references. For instance, consider the following CIF specification:

```
automaton a:
  alg string x = <string>self;

  location:
    initial;
end
```

This is transformed to the following CIF specification:

```
automaton a:
  alg string x = <string>a;

  location:
    initial;
end
```

Renaming

n/a

Size considerations

Since both references are single expression nodes, this transformation does not increase the size of the specification.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate state/event exclusion invariants

This CIF to CIF transformation eliminates state/event exclusion invariants. It replaces them with automata with self loops.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All state/event exclusion invariants are removed from the specification. They are replaced with automata with self loops.

For each named state/event exclusion invariant, a separate automaton is created. For all unnamed state/event exclusion invariants within the top level scope of the specification, a group or an automaton, as well as those in locations of that automaton, the state/event exclusion invariants are partitioned per supervisory kind. Per such partition, an automaton is created.

For invariants in the top level scope of the specification, the new automata are put in the top level scope of the specification. For invariants in a group, the new automata are put inside the group. For invariants in an automaton, the new automata are put alongside the automaton, as siblings.

For each named invariant, its new automaton gets the name of the invariant. If the named invariant is part of an automaton, or part of a location in an automaton, the name of the new automaton gets prefixed with the name of the automaton.

For unnamed invariants in a partition, the new automaton gets a name consisting of a prefix and one or more postfixes. For invariants in a group or automaton, the prefix is the name of the group or automaton. For invariants in the top level scope of the specification, an empty prefix is used. The first postfix indicates the supervisory kind of the invariants of the partition: `Plant`, `Requirement`, or `Supervisor`. For regular/kindless invariants, an empty first postfix is used. The second postfix is `StateEvtExcls`. So, for an automaton `Abc` with some plant invariants, an automaton `AbcPlantStateEvtExcls` is created.

For unnamed invariants, as well as for named invariants in automata, the name of the new automaton may conflict with existing automata or other declarations in the scope where it is placed, in which case it is renamed.

The new automata that are created inherit the supervisory kind of the state/event exclusion

invariants for which they are created. The new automata have a single nameless location, that is both initial and marked.

For each state/event exclusion invariant for which a new automaton is created, and self loop edge is created per event. The condition predicates of the invariants are used as guard, possibly after some manipulation.

For state/event exclusion invariants of the locations of automata, the condition predicate is made conditional on the location. However, this is only done for automata with at least two locations, as conditioning on an always active location is of little use. For a location `loc`, and a state/event exclusion invariant `invariant evt needs x = 1`, predicate `x = 1` is changed to `loc => x = 1`.

For state/event exclusion invariants that indicate when an event is to be disallowed/disabled, the condition predicate is inverted. The inversion is performed before the condition predicate is made conditional on the location, if applicable. For instance, for a state/event exclusion invariant `invariant x != 1 disables evt` in a location `loc`, the condition predicate `x != 1` is first inverted to `not(x != 1)`. Then it is made conditional on the location. This results in `loc => not(x != 1)`, which is then used as guard.

The alphabets of the automata are left implicit.

Consider the following example CIF specification:

```
plant automaton aut:
  controllable e1, e2, e3;
  disc int[0..10] x;

  requirement invariant e1    needs    x < 10;
  requirement invariant x = 0 disables {e1, e2};
  requirement invariant NAME: e1 needs x < 11;

  location:
    initial;

    requirement invariant x = 0 disables e3;

    edge e1 do x := x + 1;
    edge e2 do x := x - 1;
    edge e3 do x := 0;
end
```

It is transformed to the following CIF specification:

```
plant automaton aut:
  controllable e1, e2, e3;
  disc int[0..10] x;

  location:
    initial;
```

```
    edge e1 do x := x + 1;
    edge e2 do x := x - 1;
    edge e3 do x := 0;
end

requirement automaton autNAME:
  location:
    initial;
    marked;

    edge aut.e1 when aut.x < 11;
end

requirement automaton autRequirementStateEvtExcls:
  location:
    initial;
    marked;

    edge aut.e1 when aut.x < 10, not(aut.x = 0);
    edge aut.e2 when not(aut.x = 0);
    edge aut.e3 when not(aut.x = 0);
end
```

If a specification has state/event exclusion invariants for events that are not in the synchronization, send or receive alphabet of the specification, then these invariants are removed, and no automaton is created for them. This prevents the events from becoming part of the synchronization alphabet of the specification, which would mean transitions could then become possible for these events, while that was not the case before the transformation. We thus prevent the state space of the specification from changing due to such cases. If any such invariants are encountered, the transformation prints a warning.

Renaming

New automata are created. If the desired name for an automaton is already in use for some other automaton or declaration, the new automaton is given a different name. For instance, if an automaton is to be named `ControllerStateEvtExcls` and that name is already in use, it will be named `ControllerStateEvtExcls2`. If that name is also already in use, it will be named `ControllerStateEvtExcls3`, etc.

Size considerations

A constant amount of additional objects are created (e.g. an automaton, a location, a `true` initialization predicate, etc). At most one automaton is created per state/event exclusion invariant. As such, the size increase is linear in the number of state/event exclusion invariants.

Optimality

For state/event exclusion invariants that are defined in locations of automata with more than one location, this transformation makes their predicates conditional on those locations. To eliminate such location references, apply the Eliminate the use of locations in expressions CIF to CIF transformation.

This transformation generates boolean implication and inversion predicates. These are not simplified. To simply them, apply the Simplify values CIF to CIF transformation.

Annotations

This transformation does not process any specific annotations. The annotations of unnamed state/event exclusion invariants are removed. The annotations of named state/event exclusion invariants are moved to the corresponding newly created automata. The elements of the newly created automata do not have any annotations.

Eliminate state invariants

This CIF to CIF transformation eliminates state invariants. It replaces them with extra guards on edges of automata, and with extra initialization predicates in components.

Note that requirement state invariants are not eliminated by this transformation. To eliminate requirement state invariants, use data-based supervisory controller synthesis and configure it to not simplify against the requirement state invariants, so that these requirements are eliminated and incorporated in the supervisor's guards and initialization predicate.

This transformation comes in multiple variants:

Eliminate state invariants (`elim-state-invs`)

- Eliminates all kindless, plant and supervisor state invariants from the specification.

Eliminate plant state invariants (`elim-plant-state-invs`)

- Eliminates only the plant state invariants from the specification.

Eliminate supervisor state invariants (`elim-supervisor-state-invs`)

- Eliminates only the supervisor state invariants from the specification.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Synchronizing events, events that are in the alphabet of multiple automata, are not supported. To eliminate synchronization, apply the Linearize (product) or Linearize (product with merge) CIF to CIF transformation before applying this transformation.
- SVG input mappings with updates are not supported.

- Multi-assignments are not supported, e.g. `(x, y) := (y, x)` is not supported. Multiple assignments on an edge are supported.
- Partial-variable assignments are not supported.
- Time-dependent state invariants are not supported.
- State invariants that directly or indirectly depend on the values of input variables are not supported.
- State invariants that directly or indirectly contain an automaton ('self') reference are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation
- Remove CIF/SVG declarations

Implementation details

Depending on the variant of the transformation that is used, all invariants or only some of them are eliminated.

For each invariant that is eliminated, an extra initialization predicate and extra guards may be added to the specification. The state invariant itself is removed from the specification. If a removed state invariant has a name, it is lost.

Invariants in locations are adapted first, before they are eliminated. If the invariant is in a location of an automaton with two or more locations, its predicate is made conditional on being in the location. For instance, for a state invariant `invariant x < 3` in a location `loc`, the state invariant becomes `invariant loc => (x < 3)`. Regardless of the number of locations of the automaton, the state invariant is moved out of the location, and into the automaton.

For each state invariant, the predicate of the state invariant is added as an initialization predicate to the component where the state invariant is located. In order to not unnecessarily add initialization predicates to components, some steps are taken to simplify the added initialization predicate. The initial values of discrete variables and continuous variables are filled in, if it can be statically determined what their single initial value is. Similarly, a location reference is replaced by `true` or `false` as appropriate, if it can be statically determined which location is the single initial location of its automaton. If discrete variables, continuous variable, or locations are referred to via algebraic variables or derivatives of continuous variables, these algebraic variables and derivatives are inlined before the initial values are filled in. If the predicate, after filling in initial values of state objects, is trivially `true` (as can be statically determined), then no initialization predicate is added to the component. In all other cases, the predicate is simplified as much as possible before it

is added to the component.

For each state invariant, a single extra guard is added to each edge of each automaton. The added guard consists of the state invariant's predicate, where each updated variable is replaced by its new value expressions, according to the updates of the edge. Furthermore, references to the locations of that same automaton are replaced by `true` (for the target location of the edge) and `false` (for all other locations of the automaton). For instance, for an edge with update `x := x + 1` and state invariant `x < 5`, the added guard is `x + 1 < 5`. Similarly as for the added initialization predicates, algebraic variables and derivatives of continuous variables may be inlined, no extra guard is added if it is trivially `true`, and the added guards are simplified as much as possible. Additionally, no extra guard is added to an edge if the state invariant's predicate is not affected by the updates of the edge.

As an example, consider the following CIF specification:

```
automaton aut:
  disc int[1..10] x in any;
  invariant x >= 3;
  location:
    initial;
    edge do x := x - 1;
    edge do x := x + 1;
end

invariant aut.x <= 6;
```

Using the variant of this transformation that transforms all state invariants, it is transformed to the following CIF specification:

```
automaton aut:
  disc int[1..10] x in any;
  initial x >= 3;
  location:
    initial;
    edge when x - 1 <= 6, x - 1 >= 3 do x := x - 1;
    edge when x + 1 <= 6, x + 1 >= 3 do x := x + 1;
end
initial aut.x <= 6;
```

There are two state invariants. For each of them, an initialization predicate is added, in the same place as where the state invariant was originally located. Since `x` can has ten different possible initial values, the initialization predicates can't be simplified further. The edges get extra guards, one per state invariant.

In some cases, no extra initialization predicates or guards are needed, or the predicates get simplified. For instance, consider the following example:

```
automaton aut:
```

```
  disc int[1..10] x = 3;
  invariant x >= 3;
  location:
    initial;
    edge do x := 3;
    edge do x := 7;
end

invariant aut.x <= 6;
```

Using the variant of this transformation that transforms all state invariants, it is transformed to the following CIF specification:

```
automaton aut:
  disc int[1..10] x = 3;
  location:
    initial;
    edge do x := 3;
    edge when false do x := 7;
end
```

Both invariants are `true` in the initial state, and thus no initialization predicates are added. For the first edge, both `3 >= 3` and `3 <= 6` are trivially `true`, and thus no extra guards are added. For the second edge, `7 >= 3` is trivially `true` and thus no extra guard is added for it, while `7 <= 6` is simplified to `false` and added as extra guard.

Renaming

n/a

Size considerations

The specification get larger or smaller:

- A single state invariant may lead to both an initialization predicate being added and multiple extra guards being added.
- Invariants may be simplified, reducing their size. However, invariants may also get references to algebraic variables and derivatives of continuous variables inlined, increasing their size.

Optimality

For state invariants that are defined in locations of automata with more than one location, this transformation makes their predicates conditional on those locations. To eliminate such location

references, apply the CIF to CIF transformation to eliminate the use of locations in expressions.

Some effort is spent to simplify the added initialization predicates and guards. However, this simplification is not necessarily optimal; the predicates could potentially be simplified further.

Annotations

This transformation does not process or add any specific annotations. The annotations of eliminated state invariants are removed.

Eliminate `tau` event

This CIF to CIF transformation eliminates the `tau` event.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

This transformation introduces new events for the `tau` event, one per automaton where a `tau` event is used. The new events are named `tau_` (if that name is not already in use).

If the alphabet is explicit specified, it is extended as well.

The new events are neither controllable nor uncontrollable, and don't have a data type.

For instance, the following specification:

```
automaton p:
  event e;

  location l1:
    initial;
    edge when true goto l2;

  location l2:
    edge e, tau goto l1;
end
```

is transformed to the following specification:

```
automaton p:
  event e;
  event tau_;
```

```
  location l1:
    initial;
    edge tau_ when true goto l2;

  location l2:
    edge e, tau_ goto l1;
end
```

Renaming

The newly introduced `tau_` events are renamed to `tau_2`, `tau_3`, etc, if they conflict with other declarations with the same name, that already exist in the automata. If renaming is needed, a warning is printed to the console.

Size considerations

New events may be added, the alphabet may be extended, and an explicit event may be added to edges. Therefore, the size of the specification may increase.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations. The newly added events do not have any annotations.

Eliminate tuple field projections

This CIF to CIF transformation eliminates tuple field projections by replacing them with tuple index projections.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

Obtaining the value of a field of a tuple is called tuple projection. There are two ways to project a tuple. That is, there are two ways to indicate from which field to obtain the value. The first way is to

use the name of the field, the second way is to use the 0-based index of the field into the fields of the tuple's type. For instance:

```
const tuple(int a; real b) t = (1, 2.0);
const int x = t[a];
const int y = t[0];
```

Here, both constants `x` and `y` have value `1`, obtained from field `a` of the tuple value of constant `t`.

This transformation replaces tuple field projections (using the name of the field), by tuple index projections (using the 0-based index of the field). For instance, for the example above, the result of this transformation is:

```
const tuple(int a; real b) t = (1, 2.0);
const int x = t[0];
const int y = t[0];
```

Renaming

n/a

Size considerations

Since both field references and integer values are single expression nodes, this transformation does not increase the size of the specification.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Eliminate type declarations

This CIF to CIF transformation eliminates type declarations.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All uses of type declarations (in types of variables, types of other type declarations, etc), are replaced by the types of those type declarations. The type declarations themselves are removed. For instance, the following specification:

```
type t = int;
type u = v;
type v = tuple(t a, b);

const t x = 1;
const u y = (2, 3);
const v z = (4, 5);
```

is transformed to the following specification:

```
const int x = 1;
const tuple(int a, b) y = (2, 3);
const tuple(int a, b) z = (4, 5);
```

Renaming

n/a

Size considerations

Since type declarations are shortcuts for types, eliminating them could result in an increase of the size of the specification. Type declarations may be defined in terms of other type declarations. Therefore, in the worst case, the increase is exponential.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. As this transformation removes all type declarations from the specification, their annotations are removed as well.

Convert enumerations to constants

This CIF to CIF transformation converts enumerations to integer constants.

See also: Merge enumerations and Convert enumerations to integers.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order) prior to using this transformation:

- Eliminate component definition/instantiation

Implementation details

Enumerations, which are used as types, are replaced by integer types, with range `[0 .. n-1]`, where `n` is the number of literals of the enumeration.

Enumeration literals, which are used as values, are replaced by constants. If the used enumeration literal is the n^{th} literal in the corresponding enumeration (declaration), then a constant with value `n - 1` is used. That is, the integer value is the 0-based index of the enumeration literal into the literals of the enumeration declaration.

All enumeration declarations are replaced by constant integer declarations, one for each literal in the enumeration declaration.

For instance, the following specification:

```
group x:
  enum EX = A, B;

  alg EX x = A;
end

group y:
  enum EY = B, A;

  alg EY y = A;
end
```

is transformed to the following specification:

```
group x:
  alg int[0..1] x = A;
  const int A = 0;
  const int B = 1;
end

group y:
  alg int[0..1] y = A;
  const int B = 0;
  const int A = 1;
end
```

Renaming

This transformation itself does not perform any renaming.

If enumeration literals are renamed, this may influence value equality for *compatible* enumerations (enumerations with the same number of literals, with the same names, in the same order). Therefore, either apply this transformation before applying other transformations that perform renaming on enumeration literals (such as the Eliminate groups CIF to CIF transformation), or otherwise ensure that renaming does not result in an invalid specification.

Size considerations

The number of added constants is linear in the number of enumeration literals of the specification. Integer types are slightly larger than enumeration types, but the increase is linear. Constant references in expressions are not larger than enumeration literal references.

Optimality

n/a

Annotations

This transformation does not process or add any specific annotations. The annotations of enumerations are removed. The annotations of enumeration literals are moved to the newly created constants.

Convert enumerations to integers

This CIF to CIF transformation converts enumerations to integers.

See also: Merge enumerations and Convert enumerations to constants.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order) prior to using this transformation:

- Eliminate component definition/instantiation

Implementation details

Enumerations, which are used as types, are replaced by integer types, with range `[0 .. n-1]`, where `n` is the number of literals of the enumeration.

Enumeration literals, which are used as values, are replaced by integer values. If the used enumeration literal is the n^{th} literal in the corresponding enumeration (declaration), then the integer value is `n - 1`. That is, the integer value is the 0-based index of the enumeration literal into the literals of the enumeration declaration.

All enumeration declarations are removed.

For instance, the following specification:

```
group x:
  enum EX = A, B;

  const EX x = A;
end

group y:
  enum EY = B, A;

  const EY y = A;
end
```

is transformed to the following specification:

```
group x:
  const int[0..1] x = 0;
end

group y:
  const int[0..1] y = 1;
```

```
end
```

Renaming

This transformation itself does not perform any renaming.

If enumeration literals are renamed, this may influence value equality for *compatible* enumerations (enumerations with the same number of literals, with the same names, in the same order). Therefore, either apply this transformation before applying other transformations that perform renaming on enumeration literals (such as the Eliminate groups CIF to CIF transformation), or otherwise ensure that renaming does not result in an invalid specification.

Size considerations

Integer types and integer values are not larger than enumerations and their literals. Therefore, this transformation does not increase the size of the specification. In fact, it may even be smaller.

Optimality

n/a

Annotations

This transformation does not process or add any specific annotations. The annotations of enumerations and enumeration literals are removed.

Lift events

This CIF to CIF transformation lifts event declarations to the top level of the specification.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Groups are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation
- Eliminate groups

Implementation details

All events declared in the automata, are lifted to the top level of the specification.

In order to be able to track the origin of the events, they are named after their absolute names. For a group `x`, with a group `y`, with an event `e`, the absolute name of the event is `x.y.e`. The event is lifted to the top level of the specification, and named `x_y_e`.

For instance, this specification:

```
event e;

automaton a:
  event e;
  location;
end

group b:
  automaton c:
    event e;
    location;
  end
end
```

is transformed to the following specification:

```
event e;
event a_e;
event b_c_e;

automaton a:
  location;
end

group b:
  automaton c:
    location;
  end
end
```

Renaming

Since events of the automata are merged with the declarations of the specification, renaming may be necessary to ensure uniquely named declarations. For instance, for a group `x`, with a group `y`, with an event `e`, the absolute name of the event is `x.y.e`. The event is lifted to the top level of the specification, and renamed to `x_y_e`. However, if the specification already contains a declaration of a constant, automaton, other event, etc, named `x_y_e`, then the lifted event is renamed to `x_y_e2` instead. If that name is also already in use, it is renamed to `x_y_e3`, etc.

Whenever renaming takes place, a warning is printed to the console.

Size considerations

Events are lifted, which essentially means they are moved. The size of the specification does not increase.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. If the specification has a controller properties annotation, it is removed.

Linearize (merge)

This CIF to CIF transformation performs process-algebraic linearization, thereby eliminating parallel composition, event synchronization and channel communication.

This transformation eliminates non-determinism, resulting in specifications with smaller state spaces, at the expense of not preserving the specification's full behavior. To preserve non-determinism, preserving the specification's full behavior, at the expensive of a possible exponential blow-up of the specification in terms of number of edges, use the linearize (product) or linearize (product with merge) CIF to CIF transformation instead.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Specifications without automata are not supported.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove position information
- Eliminate component definition/instantiation
- Eliminate automaton `self` references
- Convert `switch` expressions to `if` expressions
- Eliminate equations
- Eliminate automata to string casts

Implementation details

A location pointer variable is introduced for each original automaton with at least two locations, and the use of locations in expressions is eliminated. This is mostly similar to what the Eliminate the use of locations in expressions CIF to CIF transformation does, except that for instance location pointer variables are added for all automata with at least two locations.

One new automaton, named `M`, is created. If all original automata have the same supervisory kind, the new automaton gets this kind as well. Otherwise, it gets no supervisory kind. The alphabet of this new automaton is the union of alphabets of the original automata, including the events that are used to send or receive in any of the automata.

The structure of the original model is kept intact as much as possible, to allow objects to retain their absolute identities (absolute names). Since all automata are linearized into a single automaton, the original automata are replaced by groups. Each replacement group will contain (as much as possible) all of the declarations, invariants, etc of its original automaton.

All discrete and continuous variables from the original automata are moved to the new automaton, to ensure they can be assigned by the new automaton. Discrete variables must be moved to the single new automaton as they can only be declared in automata. All continuous variables of the original automata are moved to the single new automaton, regardless of whether they are assigned by updates on edges or not. Continuous variables declared outside automata remain where they are. The moved discrete and continuous variables are renamed based on their absolute names. That is, for an automaton `a` with a discrete variable `v`, the absolute name of the constant is `a.v` and the moved variable will be named `a_v`.

One location, named `L`, is added to the new automaton. This location is both initial and marked.

All initialization and marker predicates from the locations of the original automata are merged together. They are each put in the group that replaces their respective original automaton. Similarly, the invariants of the locations of the original automata are also each moved to their corresponding groups. And the initialization and marker predicates specified in automata, as well as invariants specified in automata, are moved there as well.

For the `tau` event, a self loop is created per original `tau` edge. For all other events, the edges are merged, resulting in a single self loop for each non-`tau` event. By creating single edges per non-`tau` event, non-deterministic choice may be eliminated, ensuring that the model size of the resulting specification is near-linear compared to the model size of the original specification.

Monitors are taken into account when merging the guards of the edges, resulting in simpler guard predicates. Communication is eliminated altogether, and events no longer have data types after linearization. For edges with receives, the 'received value' is replaced (in the updates) by the 'send value'.

For instance, the following specification:

```
event e;

plant automaton p:
  disc int x = 1;

  location l1:
    initial;
    edge e when x = 1 do x := 2 goto l2;

  location l2:
    edge e when x = 2 do x := 1 goto l1;
end

plant automaton q:
  location l1:
    initial;
    edge tau goto l2;

  location l2:
    edge e goto l1;
end
```

is transformed to the following specification:

```
event e;

group p:
  enum LPE = l1, l2;

  initial M.p = l1 and true or M.p = l2 and false;
  marked M.p = l1 and false or M.p = l2 and false;
end

group q:
  enum LPE = l1, l2;

  initial M.q = l1 and true or M.q = l2 and false;
  marked M.q = l1 and false or M.q = l2 and false;
end

plant automaton M:
  alphabet e;
```

```
  disc int p_x = 1;
  disc .p.LPE p = .p.l1;
  disc .q.LPE q = .q.l1;

  location L:
    initial;
    marked;

    edge e when (p = .p.l1 and p_x = 1 or p = .p.l2 and p_x = 2) and
                (q = .q.l1 and false or q = .q.l2 and true)
           do if p = .p.l1 and p_x = 1: p_x := 2, p := .p.l2
              elif p = .p.l2 and p_x = 2: p_x := 1, p := .p.l1
              end,
              q := .q.l1;
    edge tau when q = .q.l1 do q := .q.l2;
end
```

We see that the automata `p` and `q` are replaced by groups. They contain 'location pointer enumerations' `LPE`, with literals for their original locations. They also contain the linearized initialization and marker predicates of the locations. Do note that the transformation simplifies these predicates beyond what is shown in this example, but they are shown here in full, to examplify the construction of such predicates. We further see that plant automata `p` and `q` are linearized to plant automaton `M`. Two location pointers, `p` and `q`, named after the original automata, are added. The discrete variables are moved. We have one location `L`, which is initial and marked. For event `e`, the guards and updates are linearized into a single self loop. Location pointer updates are incorporated as well. For event `tau`, the single original edge is simply included as a self loop. Within `M`, `p` refers to the location pointer variable and thus all references to objects in automaton `p` (now group `p`) use scope absolute references, e.g. `.p.l1`.

Non-determinism

If the original automata have non-determinism, this choice is eliminated as part of this transformation. Non-determinism can be present due to multiple outgoing edges for a single location, for the same event (excluding the `tau` event), with overlapping guards. Another cause of non-determinism is multiple senders or receivers that are enabled at the same time, for the same channel. In the resulting specification, the first possible transition is always taken, similar to how the simulator chooses, assuming the simulator is configured to always automatically choose the first transition. Linearization eliminates some of the non-determinism in this case, essentially choosing a specific trace through the state space. To ensure the same choices are made, events and automata are sorted in the same order for linearization and simulation.

For instance, the following specification:

```
automaton p:
  event e;
  disc int x = 0;
```

```
  location:
    initial;
    edge e when x < 5 do x := x + 1;
    edge e when x > 3 do x := x - 1;
end
```

is transformed to the following specification:

```
group p:
  event e;

  initial true;
  marked false;
end

automaton M:
  alphabet p.e;

  disc int p_x = 0;

  location L:
    initial;
    marked;

    edge p.e when true and (p_x < 5 or p_x > 3)
             do if true and p_x < 5: p_x := p_x + 1
                elif true and p_x > 3: p_x := p_x - 1
                end;
end
```

Here, we see that the edge for event `p.e` with guard `x < 5` and update `x := x + 1` is chosen to take precedence over the edge with guard `x > 3` with update `x := x - 1`. This choice is based on the original specification, where the edge with guard `x < 5` is listed before the edge with guard `x > 3`.

Related to this, are dummy updates, which are added to ensure that the correct updates are taken. For instance, the following specification:

```
automaton p:
  event e;
  disc int x = 0;
  location:
    initial;
    edge e when x >= 3;
    edge e when x < 3 do x := x + 1;
end
```

is transformed to the following specification:

```
group p:
  event e;

  initial true and true;
  marked true and false;
end

automaton M:
  alphabet p.e;

  disc int p_x = 0;

  location L:
    initial;
    marked;

    edge p.e when true and (p_x >= 3 or p_x < 3)
             do if true and p_x >= 3: p_x := p_x
                elif true and p_x < 3: p_x := p_x + 1
                end;
end
```

Here, the edge with guard `x >= 3` takes precedence over the edge with guard `x < 3`. To ensure that no updates are performed when the edge with guard `x >= 3` is chosen, a dummy update is added (reassigning the value of `M.p_x` to itself). If this update were to be omitted, the update of the other edge would instead be executed, which is undesirable.

Order

If code generation is performed on a linearized version of the specification, it may be a good idea to ensure the same order is used and the same choices are made, both in the generated code and in simulation. This 'merge' variant of the linearization transformation is compatible with the execution scheme defined by the CIF controller properties checker, ensuring consistency with simulation if the simulator's execution mode is used.

If the simulator's execution mode is not used, and assuming simulation was performed by always automatically choosing the first transition, this should still correspond to the output of linearization. The linearized edges are in the same order as the transitions are calculated by the simulator when execution mode is disabled. This is ensured by sorting events and automata in the same order for linearization and simulation. Also, as the Non-determinism section above explains, if non-determinism is eliminated, it is done in a way that preserves that order.

Code should thus be generated in the order of the linearized edges resulting from linearization. Each time the code for an edge is executed, the code should start from the top, to ensure always the first enabled transition is chosen, as prescribed by the execution scheme.

Urgency

If the original automata contain urgent locations and/or urgent edges, a discrete boolean variable `u` is added to the linearized automaton. Initially, it's value is `true`, and it must always remain so (`plant invariant u;`). We add self loops (event `tau`), with as guard the urgent locations and guards of urgent edges, such that the edge can be taken if the system is in an urgent location, or an urgent edge is enabled (guard wise). However, these edges update `u` to `false`, which violates the target location invariant, meaning we can never take these edges in a transition. Since the edge is also urgent, it means that if the edge is enabled guard wise, time may not progress, thus ensuring the urgency behavior of the original urgent locations and edges.

For instance, the following specification:

```
automaton p:
  event e;

  location l1:
    initial;
    urgent;
    edge e when true goto l2;

  location l2:
    edge e when 1 = 1 now goto l1;
end
```

is transformed to the following specification:

```
group p:
  event e;
  enum LPE = l1, l2;

  initial M.p = l1;
  marked false;
end

automaton M:
  alphabet .p.e;

  disc .p.LPE p = .p.l1;
  disc bool u = true;

  plant invariant u;

  location L:
    initial;
    marked;

    edge .p.e when p = .p.l1 and true or p = .p.l2 and 1 = 1
              do if p = .p.l1 and true: p := .p.l2
```

```
                elif p = .p.l2 and 1 = 1: p := .p.l1
                end;

    edge when p = .p.l1 or p = .p.l2 and 1 = 1 now do u := false;
end
```

So, if `M.p` (the location pointer variable for original automaton `p`) is equal to `p.l1` (the enumeration literal for original location `l1`), then the guard of the new urgent edge is enabled, and time may not progress. Similarly, if `M.p` is equal to `p.l2` and the guard `1 = 1` of the original urgent edge is enabled, the guard of the new urgent edge is enabled, and time may not progress. This correctly reflects the urgency conditions of the original specification.

To ensure that no additional event transitions are possible, the new urgent edge can never be taken, as it would update `u` to `false`, which violates `plant invariant u`.

Received values and tuple field projections

The following specification:

```
event tuple(int a, b) e;

automaton s:
  location:
    initial;
    edge e!(1, 2);
end

automaton r:
  disc int x;
  location:
    initial;
    edge e? do x := ?[a];
end
```

is transformed to the following specification:

```
event e;

group s:
  initial true and true;
  marked true and false;
end

group r:
  initial true and true;
  marked true and false;
end
```

```
automaton M:
  alphabet e;

  disc int r_x;

  location L:
    initial;
    marked;

    edge e when true and true and (true and true) do r_x := (1, 2)[0];
end
```

Observe how event `e` no longer has a data type, and the communication (send and receive) have been eliminated. In the assignment `x := ?[a]`, received value `?` has been replaced by send value `(1, 2)`. Since tuple values don't have field names, the right hand side `(1, 2)[a]` has been replaced by `(1, 2)[0]`, using the Eliminate tuple field projections CIF to CIF transformation.

Elimination of communication

Since channel communication is completely eliminated, and channels become regular events after this transformation, it is no longer possible to merge additional senders/receivers with the linearized specification. If you wish to merge another specification with additional communication partners, first perform the merging, and then the linearization.

Renaming

Since declarations are moved/merged, and new names are introduced, renaming may be necessary to ensure unique names within a single scope. If renaming is performed, a warning is printed to the console.

Size considerations

Variables are moved, so they don't increase the size of the specification.

The addition of location pointer variables increases the size of the specification, but this is linear in the number of automata.

Assignments are added for the updates to the location pointers. The increase is linear in the number of edges. The additional dummy updates that are added are linear in the number of edges as well.

Since `tau` edges are essentially just moved, they don't increase the size of the specification. That is, their size is linear in the number of original `tau` edges.

A single self loop is added for each non-`tau` event. This does increase the size of the specification. The guards are simply combined using boolean operators, and the increase is therefore linear in the number of guards. Similarly, updates are combined using 'if' updates, and the increase is also linear.

For original edges with multiple events however, the guards and updates are duplicated for each event. The duplication is linear in the number of events on those edges.

The initialization and marker predicates of locations, as well as the invariants of locations are moved. They remain linear in size related to the number of original predicates. The predicates are related to values of the location pointer variable. This size increase is linear in the number of original locations. Predicates are combined using boolean operators, leading to a size increase that is linear in the number of original predicates.

If a single received value is used multiple times in the updates of a single edge, the send value is duplicated for each of those uses. As such, the size increase is linear in the number of uses of the received value.

If multiple senders and multiple receivers (individual send/receive edges, not whole automata) are present for a certain event, then the received value is expanded to all the potential send values, with the conditions under which they are used as send value (using an 'if' expression). As such, the received value are expanded into expressions that are linear in the number of senders. If we have only a single sender or a single receiver, this is linear. If we have multiple of both, this becomes quadratic (number of senders times number of receivers).

For urgency, an additional variable and invariant are added. This is a constant increase in size. A self loop is added as well. This self loop duplicates the guards of urgent edges. It also includes predicates for the urgent locations. The size of this edge is linear in the number of urgent locations, and the guards of the urgent edges.

From the above, it should be clear that while the specification may increase in size, it is mostly linear. Practically, the size of the linearized specification is usually linear compared to the size of the input specification.

Optimality

This transformation performs some optimization of the generate expressions. However, often the generated expressions allow for additional simplification. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Currently, no effort is made by this transformation to reduce for instance the number of dummy assignments, or the number of replacements of tuple field projections by tuple index projections.

Annotations

This transformation does not process or add any specific annotations. The annotations of the original automata are moved to the enumeration declarations of the newly created location pointer enumerations, but only for automata with at least two locations. Similarly, the annotations of the

original locations are moved to the enumeration literals of the newly created location pointer enumerations, but only for automata with at least two locations. For automata with a single location, the annotations of the automata and their locations are removed. The single new automaton and its single new location do not have any annotations, nor do the linearized edges.

Linearize (product)

This CIF to CIF transformation performs process-algebraic linearization, thereby eliminating parallel composition, event synchronization and channel communication.

This transformation preserves non-determinism, preserving the specification's full behavior, at the expensive of a possible exponential blow-up of the specification in terms of number of edges. To eliminate non-determinism, resulting in specifications with smaller state spaces, at the expense of not preserving the specification's full behavior, use the linearize (merge) CIF to CIF transformation instead. To reduce the possibly-expontential blow-up, use the linearize (product with merge) CIF to CIF transformation instead.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Specifications without automata are not supported.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove position information
- Eliminate component definition/instantiation
- Eliminate automaton `self` references
- Convert `switch` expressions to `if` expressions
- Eliminate equations
- Eliminate automata to string casts

Implementation details

A location pointer variable is introduced for each original automaton with at least two locations, and the use of locations in expressions is eliminated. This is mostly similar to what the Eliminate the use of locations in expressions CIF to CIF transformation does, except that for instance location pointer variables are added for all automata with at least two locations.

One new automaton, named `M`, is created. If all original automata have the same supervisory kind,

the new automaton gets this kind as well. Otherwise, it gets no supervisory kind. The alphabet of this new automaton is the union of alphabets of the original automata, including the events that are used to send or receive in any of the automata.

The structure of the original model is kept intact as much as possible, to allow objects to retain their absolute identities (absolute names). Since all automata are linearized into a single automaton, the original automata are replaced by groups. Each replacement group will contain (as much as possible) all of the declarations, invariants, etc of its original automaton.

All discrete and continuous variables from the original automata are moved to the new automaton, to ensure they can be assigned by the new automaton. Discrete variables must be moved to the single new automaton as they can only be declared in automata. All continuous variables of the original automata are moved to the single new automaton, regardless of whether they are assigned by updates on edges or not. Continuous variables declared outside automata remain where they are. The moved discrete and continuous variables are renamed based on their absolute names. That is, for an automaton `a` with a discrete variable `v`, the absolute name of the constant is `a.v` and the moved variable will be named `a_v`.

One location, named `L`, is added to the new automaton. This location is both initial and marked.

All initialization and marker predicates from the locations of the original automata are merged together. They are each put in the group that replaces their respective original automaton. Similarly, the invariants of the locations of the original automata are also each moved to their corresponding groups. And the initialization and marker predicates specified in automata, as well as invariants specified in automata, are moved there as well.

For the `tau` event, a self loop is created per original `tau` edge. For all other events, the Cartesian product of all edges is created, combining the edges in all possible combinations. This results in self loops for all non-`tau` events, where the combination of all self loops preserves all non-deterministic choices of the original specification. Worst case, the model size of the resulting specification compared to the model size of the original specification could blow up exponentially.

Monitors are taken into account when merging the guards of the edges, resulting in simpler guard predicates. Communication is eliminated altogether, and events no longer have data types after linearization. For edges with receives, the 'received value' is replaced (in the updates) by the 'send value'.

For instance, the following specification:

```
event e;

plant automaton p:
  disc int x = 1;

  location l1:
    initial;
    edge e when x = 1 do x := 2 goto l2;

  location l2:
    edge e when x = 2 do x := 1 goto l1;
```

```
end

plant automaton q:
  location l1:
    initial;
    edge tau goto l2;

  location l2:
    edge e goto l1;
end
```

is transformed to the following specification:

```
event e;

group p:
  enum LPE = l1, l2;

  initial M.p = l1 and true or M.p = l2 and false;
  marked M.p = l1 and false or M.p = l2 and false;
end

group q:
  enum LPE = l1, l2;

  initial M.q = l1 and true or M.q = l2 and false;
  marked M.q = l1 and false or M.q = l2 and false;
end

plant automaton M:
  alphabet e;

  disc int p_x = 1;
  disc .p.LPE p = .p.l1;
  disc .q.LPE q = .q.l1;

  location L:
    initial;
    marked;

    edge e when p = .p.l1 and (p_x = 1 and q = .q.l2)
           do p_x := 2, p := .p.l2, q := .q.l1;
    edge e when p = .p.l2 and (p_x = 2 and q = .q.l2)
           do p_x := 1, p := .p.l1, q := .q.l1;
    edge tau when q = .q.l1 do q := .q.l2;
end
```

We see that the automata `p` and `q` are replaced by groups. They contain 'location pointer enumerations' `LPE`, with literals for their original locations. They also contain the linearized

initialization and marker predicates of the locations. Do note that the transformation simplifies these predicates beyond what is shown in this example, but they are shown here in full, to examplify the construction of such predicates. We further see that plant automata `p` and `q` are linearized to plant automaton `M`. Two location pointers, `p` and `q`, named after the original automata, are added. The discrete variables are moved. We have one location `L`, which is initial and marked. For event `e`, two self loop edges are added, as automaton `p` has two edges for `e`, automaton `q` has one edge for `e`, and 2 * 1 = 2. For the self loop edges, the guards and updates are combined. Location pointer updates are incorporated as well. For event `tau`, the single original edge is simply included as a self loop. Within `M`, `p` refers to the location pointer variable and thus all references to objects in automaton `p` (now group `p`) use scope absolute references, e.g. `.p.l1`.

Non-determinism

If the original automata have non-determinism, this choice is preserved as part of this transformation. Non-determinism can be present due to multiple outgoing edges for a single location, for the same event (excluding the `tau` event), with overlapping guards. Another cause of non-determinism is multiple senders or receivers that are enabled at the same time, for the same channel. In the resulting specification, all possible combinations are included as self loops. For a specification with 4 sender automata with two send edges each (8 send edges in total) and 5 receiver automata with two receive edges each (10 receive edges in total) for event `e`, and also 6 automata with two edges each that synchronize over that event (12 synchronization edges in total), the number of possible combinations for communication with synchronization is 8 * 10 * 12 = 960 possible combinations.

For instance, the following specification:

```
event e;

automaton p:
  disc int x = 0;

  location:
    initial;
    edge e when x < 5 do x := x + 1;
    edge e when x > 3 do x := x - 1;
end

automaton q:
  disc int y = 0;

  location:
    initial;
    edge e when y < 5 do y := y + 1;
    edge e when y > 3 do y := y - 1;
    edge e when y = 4 do y := y - 1;
end
```

is transformed to the following specification:

```
event e;

group p:
  initial true;
  marked false;
end

group q:
  initial true;
  marked false;
end

automaton M:
  alphabet e;

  disc int p_x = 0;
  disc int q_y = 0;

  location L:
    initial;
    marked;

    edge e when p_x < 5 and q_y < 5 do p_x := p_x + 1, q_y := q_y + 1;
    edge e when p_x < 5 and q_y > 3 do p_x := p_x + 1, q_y := q_y - 1;
    edge e when p_x < 5 and q_y = 4 do p_x := p_x + 1, q_y := q_y - 1;
    edge e when p_x > 3 and q_y < 5 do p_x := p_x - 1, q_y := q_y + 1;
    edge e when p_x > 3 and q_y > 3 do p_x := p_x - 1, q_y := q_y - 1;
    edge e when p_x > 3 and q_y = 4 do p_x := p_x - 1, q_y := q_y - 1;
end
```

Here, we see that for event `e`, automaton `p` has two edges and automaton `q` has three edges. The linearization result has 2 * 3 = 6 edges.

Order

If code generation is performed on a linearized version of the specification, it may be a good idea to ensure the same order is used and the same choices are made, both in the generated code and in simulation. This 'product' variant of the linearization transformation mostly adheres to the execution scheme defined by the CIF controller properties checker. However, since it may generate multiple linearized edges for the same event, it is better to use the 'merge' variant of the transformation instead, if strict compliance to the execution scheme is desired. This 'product' variant is therefore also not strictly compatible with the CIF simulator's execution mode.

Urgency

If the original automata contain urgent locations and/or urgent edges, a discrete boolean variable `u` is added to the linearized automaton. Initially, it's value is `true`, and it must always remain so (`plant invariant u;`). We add self loops (event `tau`), with as guard the urgent locations and guards of urgent edges, such that the edge can be taken if the system is in an urgent location, or an urgent edge is enabled (guard wise). However, these edges update `u` to `false`, which violates the target location invariant, meaning we can never take these edges in a transition. Since the edge is also urgent, it means that if the edge is enabled guard wise, time may not progress, thus ensuring the urgency behavior of the original urgent locations and edges.

For instance, the following specification:

```
automaton p:
  event e;

  location l1:
    initial;
    urgent;
    edge e when true goto l2;

  location l2:
    edge e when 1 = 1 now goto l1;
end
```

is transformed to the following specification:

```
group p:
  event e;
  enum LPE = l1, l2;

  initial M.p = l1;
  marked false;
end

automaton M:
  alphabet .p.e;

  disc .p.LPE p = .p.l1;
  disc bool u = true;

  plant invariant u;

  location L:
    initial;
    marked;

    edge .p.e when p = .p.l1 and true do p := .p.l2;
    edge .p.e when p = .p.l2 and 1 = 1 do p := .p.l1;
```

```
    edge when p = .p.l1 or (p = .p.l2 and 1 = 1) now do u := false;
end
```

So, if `M.p` (the location pointer variable for original automaton `p`) is equal to `p.l1` (the enumeration literal for original location `l1`), then the guard of the new urgent edge is enabled, and time may not progress. Similarly, if `M.p` is equal to `p.l2` and the guard `1 = 1` of the original urgent edge is enabled, the guard of the new urgent edge is enabled, and time may not progress. This correctly reflects the urgency conditions of the original specification.

To ensure that no additional event transitions are possible, the new urgent edge can never be taken, as it would update `u` to `false`, which violates `plant invariant u`.

Received values and tuple field projections

The following specification:

```
event tuple(int a, b) e;

automaton s:
  location:
    initial;
    edge e!(1, 2);
end

automaton r:
  disc int x;
  location:
    initial;
    edge e? do x := ?[a];
end
```

is transformed to the following specification:

```
event e;

group s:
  initial true and true;
  marked true and false;
end

group r:
  initial true and true;
  marked true and false;
end

automaton M:
```

```
  alphabet e;

  disc int r_x;

  location L:
    initial;
    marked;

    edge e when true do r_x := (1, 2)[0];
end
```

Observe how event `e` no longer has a data type, and the communication (send and receive) have been eliminated. In the assignment `x := ?[a]`, received value `?` has been replaced by send value `(1, 2)`. Since tuple values don't have field names, the right hand side `(1, 2)[a]` has been replaced by `(1, 2)[0]`, using the Eliminate tuple field projections CIF to CIF transformation.

Elimination of communication

Since channel communication is completely eliminated, and channels become regular events after this transformation, it is no longer possible to merge additional senders/receivers with the linearized specification. If you wish to merge another specification with additional communication partners, first perform the merging, and then the linearization.

Renaming

Since declarations are moved/merged, and new names are introduced, renaming may be necessary to ensure unique names within a single scope. If renaming is performed, a warning is printed to the console.

Size considerations

Variables are moved, so they don't increase the size of the specification.

The addition of location pointer variables increases the size of the specification, but this is linear in the number of automata.

Assignments are added for the updates to the location pointers. The increase is linear in the number of edges.

Since `tau` edges are essentially just moved, they don't increase the size of the specification. That is, their size is linear in the number of original `tau` edges.

Multiple self loops may be added for non-`tau` event, and thus may result in an exponential blow up of the number of edges.

The initialization and marker predicates of locations, as well as the invariants of locations are moved. They remain linear in size related to the number of original predicates. The predicates are related to values of the location pointer variable. This size increase is linear in the number of original locations. Predicates are combined using boolean operators, leading to a size increase that is linear in the number of original predicates.

If a single received value is used multiple times in the updates of a single edge, the send value is duplicated for each of those uses. As such, the size increase is linear in the number of uses of the received value.

For urgency, an additional variable and invariant are added. This is a constant increase in size. A self loop is added as well. This self loop duplicates the guards of urgent edges. It also includes predicates for the urgent locations. The size of this edge is linear in the number of urgent locations, and the guards of the urgent edges.

From the above, it should be clear that due to the exponential blow up of the edges, linearization results can be significantly larger than the original specifications.

Optimality

This transformation performs some optimization of the generate expressions. However, often the generated expressions allow for additional simplification. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Currently, no effort is made by this transformation to reduce for instance the number of replacements of tuple field projections by tuple index projections.

Annotations

This transformation does not process or add any specific annotations. The annotations of the original automata are moved to the enumeration declarations of the newly created location pointer enumerations, but only for automata with at least two locations. Similarly, the annotations of the original locations are moved to the enumeration literals of the newly created location pointer enumerations, but only for automata with at least two locations. For automata with a single location, the annotations of the automata and their locations are removed. The single new automaton and its single new location do not have any annotations, nor do the linearized edges.

Linearize (product with merge)

This CIF to CIF transformation performs process-algebraic linearization, thereby eliminating parallel composition, event synchronization and channel communication.

This transformation preserves non-determinism, preserving the specification's full behavior, at the expensive of a possible exponential blow-up of the specification in terms of number of edges. To eliminate non-determinism, resulting in specifications with smaller state spaces, at the expense of not preserving the specification's full behavior, use the linearize (merge) CIF to CIF transformation instead.

This transformation may introduce conditional updates (`if` updates). If you want to use the

linearized model in tools that do not support conditional updates, use the linearize (product) CIF to CIF transformation instead, at the expense of a potential increase of the possibly-exponential blow-up in the number of linearized edges.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Specifications without automata are not supported.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Remove position information
- Eliminate component definition/instantiation
- Eliminate automaton `self` references
- Convert `switch` expressions to `if` expressions
- Eliminate equations
- Eliminate automata to string casts

Implementation details

A location pointer variable is introduced for each original automaton with at least two locations, and the use of locations in expressions is eliminated. This is mostly similar to what the Eliminate the use of locations in expressions CIF to CIF transformation does, except that for instance location pointer variables are added for all automata with at least two locations.

One new automaton, named `M`, is created. If all original automata have the same supervisory kind, the new automaton gets this kind as well. Otherwise, it gets no supervisory kind. The alphabet of this new automaton is the union of alphabets of the original automata, including the events that are used to send or receive in any of the automata.

The structure of the original model is kept intact as much as possible, to allow objects to retain their absolute identities (absolute names). Since all automata are linearized into a single automaton, the original automata are replaced by groups. Each replacement group will contain (as much as possible) all of the declarations, invariants, etc of its original automaton.

All discrete and continuous variables from the original automata are moved to the new automaton, to ensure they can be assigned by the new automaton. Discrete variables must be moved to the single new automaton as they can only be declared in automata. All continuous variables of the original automata are moved to the single new automaton, regardless of whether they are assigned

by updates on edges or not. Continuous variables declared outside automata remain where they are. The moved discrete and continuous variables are renamed based on their absolute names. That is, for an automaton `a` with a discrete variable `v`, the absolute name of the constant is `a.v` and the moved variable will be named `a_v`.

One location, named `L`, is added to the new automaton. This location is both initial and marked.

All initialization and marker predicates from the locations of the original automata are merged together. They are each put in the group that replaces their respective original automaton. Similarly, the invariants of the locations of the original automata are also each moved to their corresponding groups. And the initialization and marker predicates specified in automata, as well as invariants specified in automata, are moved there as well.

For the `tau` event, a self loop is created per original `tau` edge. For all other events, the Cartesian product of all edges is created, combining the edges in all possible combinations. This results in self loops for all non-`tau` events, where the combination of all self loops preserves all non-deterministic choices of the original specification. Worst case, the model size of the resulting specification compared to the model size of the original specification could blow up exponentially.

Monitors are taken into account when merging the guards of the edges, resulting in simpler guard predicates. Communication is eliminated altogether, and events no longer have data types after linearization. For edges with receives, the 'received value' is replaced (in the updates) by the 'send value'.

Where possible, exponential blow-up is prevented by merging multiple alternatives, via which an automaton may participate in synchronization, into a single alternative. Essentially, if there is no guard overlap and thus no non-determism, it does something similar to linearize (merge) CIF to CIF transformation instead. For more information, see the section about non-determinism.

For instance, the following specification:

```
event e;

plant automaton p:
  disc int x = 1;

  location l1:
    initial;
    edge e when x = 1 do x := 2 goto l2;

  location l2:
    edge e when x = 2 do x := 1 goto l1;
end

plant automaton q:
  location l1:
    initial;
    edge tau goto l2;

  location l2:
```

```
    edge e goto l1;
end
```

is transformed to the following specification:

```
event e;

group p:
  enum LPE = l1, l2;

  initial M.p = l1 and true or M.p = l2 and false;
  marked M.p = l1 and false or M.p = l2 and false;
end

group q:
  enum LPE = l1, l2;

  initial M.q = l1 and true or M.q = l2 and false;
  marked M.q = l1 and false or M.q = l2 and false;
end

plant automaton M:
  alphabet e;

  disc int p_x = 1;
  disc .p.LPE p = .p.l1;
  disc .q.LPE q = .q.l1;

  location L:
    initial;
    marked;

    edge e when (p = .p.l1 and p_x = 1) or (p = .p.l2 and p_x = 2)
                and q = .q.l2
           do if   p = .p.l1: p_x := 2, p := .p.l2
              elif p = .p.l2: p_x := 1, p := .p.l1
              end,
              q := .q.l1;
    edge tau when q = .q.l1 do q := .q.l2;
end
```

We see that the automata `p` and `q` are replaced by groups. They contain 'location pointer enumerations' `LPE`, with literals for their original locations. They also contain the linearized initialization and marker predicates of the locations. Do note that the transformation simplifies these predicates beyond what is shown in this example, but they are shown here in full, to examplify the construction of such predicates. We further see that plant automata `p` and `q` are linearized to plant automaton `M`. Two location pointers, `p` and `q`, named after the original automata, are added. The discrete variables are moved. We have one location `L`, which is initial and marked. For event `e`, one self loop edge is added. In principle, automaton `p` has two edges for `e` and

automaton `q` has one edge for `e`, leading to 2 * 1 = 2 edges in the product. However, in this case, these two edges can be merged into a single edge, as the originate from different source locations and are thus never enabled at the same time (there is no non-determinism). The guards of the original edges are combined, as are the updates. Location pointer updates are incorporated as well. For event `tau`, the single original edge is simply included as a self loop. Within `M`, `p` refers to the location pointer variable and thus all references to objects in automaton `p` (now group `p`) use scope absolute references, e.g. `.p.l1`.

Non-determinism

If the original automata have non-determinism, this choice is preserved as part of this transformation. Non-determinism can be present due to multiple outgoing edges for a single location, for the same event (excluding the `tau` event), with overlapping guards. Another cause of non-determinism is multiple senders or receivers that are enabled at the same time, for the same channel. In the resulting specification, in principle all possible combinations are included as self loops, although some optimizations are performed as we will discuss below. In principle, for a specification with 4 sender automata with two send edges each (8 send edges in total) and 5 receiver automata with two receive edges each (10 receive edges in total) for event `e`, and also 6 automata with two edges each that synchronize over that event (12 synchronization edges in total), the number of possible combinations for communication with synchronization is 8 * 10 * 12 = 960 possible combinations.

For instance, the following specification:

```
event e;

automaton p:
  disc int x = 0;

  location:
    initial;
    edge e when x < 5 do x := x + 1;
    edge e when x > 3 do x := x - 1;
end

automaton q:
  disc int y = 0;

  location:
    initial;
    edge e when y < 5 do y := y + 1;
    edge e when y > 3 do y := y - 1;
    edge e when y = 4 do y := y - 1;
end
```

is transformed to the following specification:

```
event e;

group p:
  initial true;
  marked false;
end

group q:
  initial true;
  marked false;
end

automaton M:
  alphabet e;

  disc int p_x = 0;
  disc int q_y = 0;

  location L:
    initial;
    marked;

    edge e when p_x < 5 and q_y < 5 do p_x := p_x + 1, q_y := q_y + 1;
    edge e when p_x < 5 and q_y > 3 do p_x := p_x + 1, q_y := q_y - 1;
    edge e when p_x < 5 and q_y = 4 do p_x := p_x + 1, q_y := q_y - 1;
    edge e when p_x > 3 and q_y < 5 do p_x := p_x - 1, q_y := q_y + 1;
    edge e when p_x > 3 and q_y > 3 do p_x := p_x - 1, q_y := q_y - 1;
    edge e when p_x > 3 and q_y = 4 do p_x := p_x - 1, q_y := q_y - 1;
end
```

Here, we see that for event `e`, automaton `p` has two edges and automaton `q` has three edges. The linearization result has 2 * 3 = 6 edges.

In certain cases, when it can be statically determined that there is no guard overlap, and thus no non-determinism, edges are merged to prevent blow-ups in the number of linearized edges. Concretely, this transformation merges linearized edges in the following cases:

- *Merge of single explicit edge in a location with its implicit monitor edge*:

  An implicit monitor edge, by definition, never has guard overlap with the other edges in the same location. Hence, there is no non-determinism, and we can safely merge the two edges.

  Concretely, if an event is monitored in an automaton, and a location of that automaton has exactly one edge for the event, and the edge has a guard, which is not explicitly `true`, then the implicit monitor edge is merged into the explicit edge, combining the two edges into one edge. Then, there is only one way for the automaton to participate in synchronization for that event, from that location, rather than there being two ways for it to participate. If an edge is labeled multiple times with the same event, then this counts as the location having multiple edges for the event, and the merge is not performed.

As an example, consider an automaton `a` that monitors event `e`, and one of its locations `g` with only the edge:

```
edge e when x > 1 do x := x + 1 goto h;
```

When this explicit edge is combined with the implicit monitor edge, it becomes always enabled in the source location. The updates, so both the explicit one and the one introduced by linearization to update the location pointer variable, are made conditional on the original guard of the explicit edge. This results in a single linearized self-loop edge:

```
edge e when a = g do if x > 1: x := x + 1, a := h end;
```

- *Merge of single participations from different source locations*:

  Edges from different source locations of the same automaton can never be enabled at the same time, as the automaton can only be in one location at a time. Hence, there is no non-determinism, and we can safe merge those edges.

  Concretely, this transformation combines edges for the same event, with different source locations, from an automaton that synchronizes over the event, if they are the only synchronization options in their source locations. This merges two or more edges into a single edge. If a location has multiple explicit edges for the event, or it has a single edge labeled multiple times with that event, then the location has multiple ways to participate in synchronization and these edges are not merged.

  As an example, consider an automaton `a` that synchronizes over event `e` and monitors that event:

```
event e;

automaton a:
  monitor e;
  disc int x;

  location g:
    initial;
    edge e when x > 1 do x := x + 1 goto h;

  location h:
    edge e goto i;

  location i:
    edge e when x >  1 goto g;
    edge e when x <= 1 goto h;
end
```

  By the merging of the explicit edge in location `g` with the implicit monitor edge for that location,

as explained above, location `g` has only one way to participate in the synchronization for event `e`. Location `h` only has one explicit edge for event `e`, and does not get an implicit monitor edge as the explicit edge is always enabled. Hence, location `h` also has only one way to participate in the synchronization for event `e`. Location `i` has two explicit edges, and gets an implicit monitor edge, and can thus participate in three ways in the synchronization for event `e`. We merge the participations for locations `g` and `h`, as they both have only a single participation option. We then get four linearized edges, one for the merged participation of locations `g` and `h`, and three for the participations of location `i`. The linearized edges are as follows (simplified a bit for clarity):

```
edge e when a = g or a = h do if   a = g and a_x > 1: a_x := a_x + 1, a := h
                              elif a = h:                           a := i
                              end;
edge e when a = i and a_x >  1 do a := g;
edge e when a = i and a_x <= 1 do a := h;
edge e when a = i and not(a_x > 1) and not(a_x <= 1);
```

In general, if an automaton has `n` ways to participate in synchronization, and there are `m` such automata, then the number of linearized edges is n^m. There is thus an exponential blow-up. If the `n` participation options can be merged into a single participation option, then the number of linearized edges is $1^m = 1$ instead, preventing the exponential blow-up altogether. While this transformation prevents such blow-ups in some common cases, it does not prevent all such blow-ups, such as when there are two edges for the same event from the same source location.

Order

If code generation is performed on a linearized version of the specification, it may be a good idea to ensure the same order is used and the same choices are made, both in the generated code and in simulation. This 'product with merge' variant of the linearization transformation mostly adheres to the execution scheme defined by the CIF controller properties checker. However, since it may generate multiple linearized edges for the same event, it is better to use the 'merge' variant of the transformation instead, if strict compliance to the execution scheme is desired. This 'product with merge' variant is therefore also not strictly compatible with the CIF simulator's execution mode.

Urgency

If the original automata contain urgent locations and/or urgent edges, a discrete boolean variable `u` is added to the linearized automaton. Initially, it's value is `true`, and it must always remain so (`plant invariant u;`). We add self loops (event `tau`), with as guard the urgent locations and guards of urgent edges, such that the edge can be taken if the system is in an urgent location, or an urgent edge is enabled (guard wise). However, these edges update `u` to `false`, which violates the target location invariant, meaning we can never take these edges in a transition. Since the edge is also urgent, it means that if the edge is enabled guard wise, time may not progress, thus ensuring the urgency behavior of the original urgent locations and edges.

For instance, the following specification:

```
automaton p:
  event e;

  location l1:
    initial;
    urgent;
    edge e when true goto l2;

  location l2:
    edge e when 1 = 1 now goto l1;
end
```

is transformed to the following specification:

```
group p:
  event e;
  enum LPE = l1, l2;

  initial M.p = l1;
  marked false;
end

automaton M:
  alphabet .p.e;

  disc .p.LPE p = .p.l1;
  disc bool u = true;

  plant invariant u;

  location L:
    initial;
    marked;

    edge .p.e when p = .p.l1 or (p = .p.l2 and 1 = 1) do if   p = .p.l1: p := .p.l2
                                                          elif p = .p.l2: p := .p.l1
                                                          end;

    edge when p = .p.l1 or (p = .p.l2 and 1 = 1) now do u := false;
end
```

So, if `M.p` (the location pointer variable for original automaton `p`) is equal to `p.l1` (the enumeration literal for original location `l1`), then the guard of the new urgent edge is enabled, and time may not progress. Similarly, if `M.p` is equal to `p.l2` and the guard `1 = 1` of the original urgent edge is enabled, the guard of the new urgent edge is enabled, and time may not progress. This correctly reflects the urgency conditions of the original specification.

To ensure that no additional event transitions are possible, the new urgent edge can never be taken, as it would update `u` to `false`, which violates `plant invariant u`.

Received values and tuple field projections

The following specification:

```
event tuple(int a, b) e;

automaton s:
  location:
    initial;
    edge e!(1, 2);
end

automaton r:
  disc int x;
  location:
    initial;
    edge e? do x := ?[a];
end
```

is transformed to the following specification:

```
event e;

group s:
  initial true and true;
  marked true and false;
end

group r:
  initial true and true;
  marked true and false;
end

automaton M:
  alphabet e;

  disc int r_x;

  location L:
    initial;
    marked;

    edge e when true do r_x := (1, 2)[0];
end
```

Observe how event `e` no longer has a data type, and the communication (send and receive) have been eliminated. In the assignment `x := ?[a]`, received value `?` has been replaced by send value `(1, 2)`. Since tuple values don't have field names, the right hand side `(1, 2)[a]` has been replaced by `(1, 2)[0]`, using the Eliminate tuple field projections CIF to CIF transformation.

Elimination of communication

Since channel communication is completely eliminated, and channels become regular events after this transformation, it is no longer possible to merge additional senders/receivers with the linearized specification. If you wish to merge another specification with additional communication partners, first perform the merging, and then the linearization.

Renaming

Since declarations are moved/merged, and new names are introduced, renaming may be necessary to ensure unique names within a single scope. If renaming is performed, a warning is printed to the console.

Size considerations

Variables are moved, so they don't increase the size of the specification.

The addition of location pointer variables increases the size of the specification, but this is linear in the number of automata.

Assignments are added for the updates to the location pointers. The increase is linear in the number of edges.

Since `tau` edges are essentially just moved, they don't increase the size of the specification. That is, their size is linear in the number of original `tau` edges.

Multiple self loops may be added for non-`tau` event, and thus may result in an exponential blow up of the number of edges.

The initialization and marker predicates of locations, as well as the invariants of locations are moved. They remain linear in size related to the number of original predicates. The predicates are related to values of the location pointer variable. This size increase is linear in the number of original locations. Predicates are combined using boolean operators, leading to a size increase that is linear in the number of original predicates.

If a single received value is used multiple times in the updates of a single edge, the send value is duplicated for each of those uses. As such, the size increase is linear in the number of uses of the received value.

For urgency, an additional variable and invariant are added. This is a constant increase in size. A

self loop is added as well. This self loop duplicates the guards of urgent edges. It also includes predicates for the urgent locations. The size of this edge is linear in the number of urgent locations, and the guards of the urgent edges.

From the above, it should be clear that due to the exponential blow up of the edges, linearization results can be significantly larger than the original specifications.

Optimality

This transformation performs some optimization of the generate expressions. However, often the generated expressions allow for additional simplification. To further simplify the result, apply additional CIF to CIF transformations, such as Simplify values.

Currently, no effort is made by this transformation to reduce for instance the number of replacements of tuple field projections by tuple index projections.

Annotations

This transformation does not process or add any specific annotations. The annotations of the original automata are moved to the enumeration declarations of the newly created location pointer enumerations, but only for automata with at least two locations. Similarly, the annotations of the original locations are moved to the enumeration literals of the newly created location pointer enumerations, but only for automata with at least two locations. For automata with a single location, the annotations of the automata and their locations are removed. The single new automaton and its single new location do not have any annotations, nor do the linearized edges.

Merge enumerations

This CIF to CIF transformation merges all enumerations together to a single enumeration.

See also: Convert enumerations to integers and Convert enumerations to constants.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order) prior to using this transformation:

- Eliminate component definition/instantiation

The following CIF to CIF transformations are automatically applied as preprocessing (in the given

order), to increase the subset of CIF specifications that can be transformed:

- Add default initial values

Implementation details

If there are any enumerations in the specification, a single new enumeration is created. This enumeration is named `E`. All literals of the original enumerations are added to this new enumeration. Obviously, duplicate literals are only included once. The literals are added in sorted order.

The original enumerations are removed.

For instance, the following specification:

```
enum A = B, C;

automaton p:
  enum A = C, B;
  enum X = D, C;
  disc A a = C;
  disc X x = D;
  location:
    initial;
end
```

is transformed to the following specification:

```
enum E = B, C, D;

automaton p:
  disc E a = C;
  disc E x = D;
  location:
    initial;
end
```

Renaming

The newly created enumeration `E` and its literals, may conflict with declarations already present in the specification. If this is the case, they are renamed. For instance, if `E` is already in use, `E2`, or `E3`, etc, is used instead. If renaming is performed, a warning is printed to the console.

In general, renaming of enumeration literals may influence value equality for *compatible* enumerations (enumerations with the same number of literals, with the same names, in the same

order). However, since the resulting specification has at most one enumeration, there are no multiple enumerations, and thus compatibility is not an issue (since the enumeration is always compatible with itself).

Default initial values

Consider the following specification:

```
group g1:
  enum e1 = A, B;
end

group g2:
  enum e2 = B, A;
end

automaton p:
  disc g1.e1 v1;
  disc g2.e2 v2;

  location:
    initial;
end
```

If we apply merging of enumerations directly, we get the following:

```
enum E = A, B;

group g1:
end

group g2:
end

automaton p:
  disc E v1;
  disc E v2;

  location:
    initial;
end
```

In the original specification, `v1` had value `A`, and `v2` had value `B`. After the merging of the enumerations, both `v1` and `v2` have implicit initial value `A`. Thus, they had different initial values beforehand, and the same initial values afterward. To solve this problem, the Add default initial values CIF to CIF transformation is automatically applied as preprocessing before the actual

elimination of the enumerations, to ensure that the explicit initial values are properly transformed.

Switch expressions

Consider the following specification:

```
enum E1 = A, B;
enum E2 = C, D;

input E1 i;
invariant switch i:
            case A: true
            case B: false
          end;
```

After merging the enumerations, the single merged enumeration has four literals (`A`, `B`, `C` and `D`). The `switch` expression is then incomplete, as it does not cover the extra literals from `E2` (`C` and `D`) that got merged with `E1`. To ensure that `switch` expressions remain complete and valid, all `switch` expressions that switch over a value that has a type that includes an enumeration type, but do not have an `else`, are modified. Their last `case` is converted to an `else`, removing the key of the `case` in the process. For the example above, this results in:

```
enum E = A, B, C, D;

input E i;
invariant switch i:
            case A: true
            else    false
          end;
```

Size considerations

Since enumerations are merged, duplicate literals become a single new literal. Therefore, this transformation may decrease the size of the specification.

The added default initial values may significantly increase the size of the specification.

Removing the key of the last `case` of a `switch` expressions may slightly decrease the size of the specification.

Optimality

n/a

Annotations

This transformation does not process or add any specific annotations. The annotations of enumerations are removed. The annotations of enumeration literals are moved to the merged literals.

Push print file declarations into print declarations

This CIF to CIF transformation pushes print file declarations inwards into the print declarations.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All print file declarations specified in the top level scope of the specification, in groups, and in automata, are removed. The print declarations to which they applied, each get a local print file declaration instead. If they already had a local print file declaration, the print declaration is not changed. Print declarations to which no print file declaration applies, get the default target `":stdout"`.

For instance, this specification:

```
printfile "out1.txt";

print 5;
print 6 file "out2.txt";

group g:
  printfile "out3.txt";

  print 7;
  print 8;
  print 9 file "out4.txt";
```

```
  group h:
    printfile "out5.txt";
  end

  group i:
    print 10;
  end
end
```

is transformed to the following specification:

```
print 5 file "out1.txt";
print 6 file "out2.txt";

group g:
  print 7 file "out3.txt";
  print 8 file "out3.txt";
  print 9 file "out4.txt";

  group h:
  end

  group i:
    print 10 file "out3.txt";
  end
end
```

The print declaration that prints `5` is declared in the top level scope of the specification, and does not have a local print file declaration. As such, the print file declaration from the top level scope (`out1.txt`) is pushed inwards into the print declaration. The print declaration that prints `6` is declared in the same scope, but already has a local print file declaration (`out2.txt`), and is thus not changed.

Group `g` also has an print file declaration (`out3.txt`). This is pushed inwards to the print declarations that print `7` and `8`, as they are declared in the same scope, and don't have local print file declarations. The print declaration that prints `9` is not changed, as it already has a local print file declaration (`out4.txt`).

The print file declaration (`out5.txt`) from group `g.h` does not apply to any print declarations, and thus gets lost.

Group `g.i` does not have an print file declaration, so the print declaration that prints `10`, which is declared in that group, gets the print file declaration from group `g` (the parent scope of group `g.i`) instead.

Renaming

n/a

Size considerations

As print file declarations may apply to multiple print declarations, this transformation may increase the number of (local) print file declarations in the specification. The number of print file declarations after the transformation is at most the number of print declarations, and as such the increase is at most linear in the number of print declarations as well.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Push SVG file declarations into other CIF/SVG declarations

This CIF to CIF transformation pushes SVG file declarations inwards into the other CIF/SVG declarations.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All SVG file declarations specified in the top level scope of the specification, in groups, and in automata, are removed. The other CIF/SVG declarations to which they apply, each get a local SVG file declaration instead. If they already had a local SVG file declaration, the CIF/SVG declaration is not changed.

For instance, this specification:

```
svgfile "file1.svg";
```

```
svgout id "a" attr "b" value 5;

svgout id "c" attr "d" value 6 file "file2.svg";

group g:
  svgfile "file3.svg";

  svgout id "e" attr "f" value 7;

  svgout id "g" attr "h" value 8;

  svgout id "i" attr "j" value 9 file "file4.svg";

  group h:
    svgfile "file5.svg";
  end

  group i:
    svgout id "k" attr "l" value 10;
  end
end
```

is transformed to the following specification:

```
svgout id "a" attr "b" value 5 file "file1.svg";

svgout id "c" attr "d" value 6 file "file2.svg";

group g:
  svgout id "e" attr "f" value 7 file "file3.svg";

  svgout id "g" attr "h" value 8 file "file3.svg";

  svgout id "i" attr "j" value 9 file "file4.svg";

  group h:
  end

  group i:
    svgout id "k" attr "l" value 10 file "file3.svg";
  end
end
```

The mapping for the SVG element with id `a` is declared in the top level scope of the specification, and does not have a local SVG file declaration. As such, the SVG file declaration from the top level scope (`file1.svg`) is pushed inwards into the mapping. The mapping for the SVG element with id `b` is declared in the same scope, but already has a local SVG file declaration (`file2.svg`), and is thus not changed.

Group `g` also has an SVG file declaration (`file3.svg`). This is pushed inwards to the mappings for ids `e` and `g`, as they are declared in the same scope, and don't have local SVG file declarations. The mapping for SVG element id `i` is not changed, as it already has a local SVG file declaration (`file4.svg`).

The SVG file declaration (`file5.svg`) from group `g.h` does not apply to any mappings, and thus gets lost.

Group `g.i` does not have an SVG file declaration, so the mapping for SVG element id `k`, which is declared in that group, gets the SVG file declaration from group `g` (the parent scope of group `g.i`) instead.

Renaming

n/a

Size considerations

As SVG file declarations may apply to multiple other CIF/SVG declarations, this transformation may increase the number of SVG file declarations in the specification. The number of SVG file declarations after the transformation is at most the number of 'other' CIF/SVG declarations, and as such the increase is at most linear in the number of 'other' CIF/SVG declarations as well.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Relabel supervisors as plants

This CIF to CIF transformation relabels supervisors as plants.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

This CIF to CIF transformation relabels:

- Supervisor automaton definitions as plant automaton definitions.

- Supervisor automata as plant automata.
- Supervisor invariants as plant invariants.

For instance, consider the following partial CIF specification:

```
supervisor button:
  location off:
    initial;
    marked;
    edge u_on goto on;
  location on:
    edge u_off goto off;
end

supervisor invariant actuator.c_on needs button.pushed;
```

It gets transformed to:

```
plant button:
  location off:
    initial;
    marked;
    edge u_on goto on;
  location on:
    edge u_off goto off;
end

plant invariant actuator.c_on needs button.pushed;
```

Renaming

n/a

Size considerations

Automata and invariants may get a different supervisory kind. This does not change the size of the specification.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Remove annotations

This CIF to CIF transformation removes annotations from the specification. This may reduce the

size of the specification. It may also lead to the specification being supported by more tools, as the annotations may use features of the CIF language that certain tools don't support.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

All annotations are removed from the specification.

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation removes all annotations.

Remove CIF/SVG declarations

This CIF to CIF transformation removes CIF/SVG declarations from the specification. This may reduce the size of the specification. It may also lead to the specification being supported by more tools, as the CIF/SVG declarations may use features of the CIF language that certain tools don't support.

This CIF to CIF transformation removes only CIF/SVG declarations. The Remove I/O declarations CIF to CIF transformation removes them as well, but additionally also removes the other I/O declarations.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

All CIF/SVG declarations are removed from the specification.

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Remove I/O declarations

This CIF to CIF transformation removes I/O declarations from the specification. This may reduce the size of the specification. It may also lead to the specification being supported by more tools, as the I/O declarations may use features of the CIF language that certain tools don't support.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

All I/O declarations are removed from the specification. I/O declarations include:

- Print declarations
- Print file declarations
- CIF/SVG declarations

This transformation removes all I/O declarations. To only remove certain kinds of I/O declarations, see the following CIF to CIF transformations:

- Remove print declarations
- Remove CIF/SVG declarations

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Remove print declarations

This CIF to CIF transformation removes print and print file declarations from the specification. This may reduce the size of the specification. It may also lead to the specification being supported by more tools, as the print and print file declarations may use features of the CIF language that certain tools don't support.

This CIF to CIF transformation removes only print and print file declarations. The Remove I/O declarations CIF to CIF transformation removes them as well, but additionally also removes the other I/O declarations.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

All print and print file declarations are removed from the specification.

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Remove position information

This CIF to CIF transformation removes position information from the internal representation of the specification.

Supported specifications

This transformation supports all CIF specifications.

Preprocessing

n/a

Implementation details

When performing operations on CIF specifications, the specification is first read from disk, and stored into memory. The internal representation stores position information with all objects (such as declarations, types, expression nodes, automata, locations, etc). The position information of an object includes information about the location of the object in the original textual source. This information is used during validation of the specification (i.e. type checking), when reporting errors, to indicate where in the original specification the error is found.

After type checking, this information is no longer used. To make sure that subsequent operations, such as CIF to CIF transformations, don't have to process the position information, this transformation can be applied to rid the specification of all position information. Applying this transformation may increase the performance of subsequent transformations.

Applying this transformation has no effect on the actual contents of the specification, and does not change the effect of subsequent transformation. That is, it may affect the performance of subsequent transformations, but has no effect on the result of the subsequent transformations, as they are written to disk.

Renaming

n/a

Size considerations

Since position information is removed from the specification, this transformation has an effect on the size of the specification as it is represented in memory. However, this transformation has no effect on the size of the specification, as far as actual content (which affects the meaning of the specification) is concerned.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Remove requirements

This CIF to CIF transformation removes requirements from the specification. By removing all requirements from a CIF specification with both plants and requirements, only the plants remain. The resulting plants can then be merged with a supervisor or other controller.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.
- Using (referring to) requirement automata or declarations from requirement automata from outside requirement automata is not supported.
- Kindless, plant, or supervisor invariants in locations of a requirement automaton are not supported.

Specifications where requirement automata or declarations declared in requirement automata are used (referred to) from outside requirement automata are not supported. By removing the requirement automata, those uses (references) become invalid, as the automata or declarations that are referenced no longer exist. For instance, consider:

```
alg int x = switch req: case l1: 1 else 2 end;

requirement automaton req:
  location l1:
    initial;
    edge tau goto l2;

  location l2;
```

```
end
```

When requirement automaton `req` is removed, the `switch` expression can no longer refer to that automaton, or to its locations. Removing the requirements from this CIF specification is not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

All types of requirements are removed:

- Requirement automata
- Requirement invariants
- Reachability requirement annotations

Requirement invariants are removed from the top level scope of the specification, all groups and automata, and all locations of automata. For instance, consider the following CIF specification:

```
controllable add;

plant automaton buffer:
  disc int x;

  plant invariant 0 <= x and x <= 5;
  requirement invariant x < 5;

  location:
    initial;
    plant invariant x >= 0;
    edge add do x := x + 1;
end

requirement automaton not2:
  location:
    initial;
    requirement invariant buffer.x >= 0;
    edge add when buffer.x != 2;
end
```

The result after this transformation is:

```
controllable add;

plant automaton buffer:
  disc int x;

  plant invariant 0 <= x and x <= 5;

  location:
    initial;
    plant invariant x >= 0;
    edge add do x := x + 1;
end
```

Similarly, reachability requirement annotations are removed from the top-level scope of the specification as well as from all components (automata and groups).

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process or add any annotations.

All reachability requirement annotations are removed. Furthermore, if requirement automata have annotations, or if elements in requirement automata have annotations, removing these requirement automata removes those annotations as well. Similarly, if requirement invariants are removed, then so are their annotations.

If the specification has a controller properties annotation, it is removed.

Remove unused algebraic variables

This CIF to CIF transformation removes unused algebraic variables and their equations from the specification. As these removed algebraic variables are defined but not used, removing them does not alter the behavior modeled by the CIF specification.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

An algebraic variable is considered to be *unused* in a CIF specification if it only occurs as an algebraic variable definition.

```
alg bool var1 = false;
alg bool var2 = true;

plant A:
  uncontrollable a;
  location:
    initial;
    edge a when var1;
end
```

As algebraic variable `var2` is defined but not mentioned elsewhere in the specification it is removed. Algebraic variable `var1` is mentioned in an edge guard. It is therefore considered *used* and preserved as you can see in the resulting specification:

```
alg bool var1 = false;

plant A:
  uncontrollable a;
  location:
    initial;
    edge a when var1;
end
```

As algebraic variables may be used to define other algebraic variables, removal of an algebraic variable may trigger more removals. For instance, consider the following specification:

```
alg bool var1 = false;
alg bool var2 = not var1;
alg bool var3 = var2;
```

```
plant A:
  uncontrollable a;
  location:
    initial;
    edge a when var1;
end
```

Algebraic variable `var3` is only defined but not mentioned elsewhere. It is thus considered *unused* and removed. After the removal of `var3`, algebraic variable `var2` is not mentioned elsewhere anymore. It is thus now considered *unused* as well and also removed. Algebraic variable `var1` is considered *used* as before and not removed.

The result is therefore:

```
alg bool var1 = false;

plant A:
  uncontrollable a;
  location:
    initial;
    edge a when var1;
end
```

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. As this transformation removes all unused algebraic variables from the specification, their annotations are removed as well.

Remove unused events

This CIF to CIF transformation removes unused events from the specification. As these removed events are defined but not used, removing them does not alter the behavior modeled by the CIF specification.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

An event is considered to be *unused* in a CIF specification if it only occurs as an event declaration.

```
event a;
event b;

automaton A:
  location:
    initial;
    edge a;
end
```

As event `b` is defined but not mentioned elsewhere in the specification it is removed. Event `a` is mentioned on an edge. It is therefore considered *used* and preserved as you can see in the resulting specification:

```
event a;

automaton A:
  location:
    initial;
    edge a;
end
```

Take note that the implicit alphabet of an automaton is defined by the events on the edges and not by the declared events, see the lesson on the alphabet. For instance, consider the following specification:

```
automaton A:
  event a;
  event b;
  location:
    initial;
    edge a;
```

```
end
```

Event `b` is only defined but not mentioned elsewhere. Therefore, it is not part of the implicit alphabet of automaton `A`. It is thus considered *unused* and removed.

The result is therefore:

```
automaton A:
  event a;
  location:
    initial;
    edge a;
end
```

Renaming

n/a

Size considerations

Since information is removed from the specification, the specification can only become smaller as a result of this transformation.

Optimality

n/a

Annotations

This transformation does not process or add any annotations. If unused events have annotations, then by removing the unused events their annotations are removed as well.

Simplify others

This CIF to CIF transformation simplifies CIF specifications, by applying the following non-value related simplifications:

- Duplicate monitor events of automata are removed.
- Duplicate events in the alphabet are removed.
- Duplicate events on edges are removed, with the exception of duplicate sends for the same event, as they may send different values.
- Self-loops with a target location are simplified by removing the target location.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

n/a

Renaming

n/a

Size considerations

This transformation tries to simplify the specification, possibly reducing its size.

Optimality

n/a

Annotations

This transformation does not process, add, or remove any annotations.

Simplify values

This CIF to CIF transformation simplifies CIF specifications, by applying the following value-related simplifications:

- Evaluation of constant (sub-)expressions. For instance, `1 + 1` is simplified to `2`.
- Simplification of short-circuit boolean binary operators. For instance, `true and x` is simplified to `x`.
- Simplification of boolean unary operators. For instance, `not not x` is simplified to `x`.
- Removal of default values. For instance, `true` state invariants are removed.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

The following is a complete list of the short-circuit boolean binary operators that are simplified (with `x` an arbitrary boolean expression):

Original	Simplified
`true and x`	`x`
`x and true`	`x`
`false and x`	`false`
`x and false`	`false`
`true or x`	`true`
`x or true`	`true`
`false or x`	`x`
`x or false`	`x`
`true => x`	`x`
`x => true`	`true`
`false => x`	`true`
`x => false`	`not x`
`true <=> x`	`x`
`x <=> true`	`x`
`false <=> x`	`not x`
`x <=> false`	`not x`

The following is a complete list of the boolean unary operators that are simplified (with `x` and `y` two arbitrary expressions):

Original	Simplified
`not not x`	`x`
`not(x < y)`	`x >= y`
`not(x <= y)`	`x > y`
`not(x = y)`	`x != y`
`not(x != y)`	`x = y`

Original	Simplified
`not(x > y)`	`x ⇐ y`
`not(x >= y)`	`x < y`
`not(x => y)`	`x and not y`

Default values are removed in the following places:

- Initialization predicates of components (including automata) and locations.
- Marker predicates of components (including automata) and locations.
- Invariants of components (including automata) and locations.
- Guards of edges.
- Guards of 'if' and 'elif' updates on edges and in SVG input mappings.
- Guards of 'if' and 'elif' expressions.

Predicates that are trivially equal to the default value are removed. If a trivial value is found that is equal to the negation of the default value, the entire feature gets the non-default value. For instance, for guards on edges, `true`, `1 = 1`, etc, are removed as they are all trivially `true` (the default for guards). If however, `false`, `1 = 2`, `1 != 1`, or any other trivially `false` guards is found, all guards on that edge are removed, and a single `false` guard is added.

Renaming

n/a

Size considerations

This transformation tries to simplify the specification, possibly reducing its size.

All uses of constants lead to constant (sub-)expressions, and they are evaluated. This leads to constants being inlined. For constants with large literal values, this may significantly increase the size of the specification, especially if the constant is used more than once. For information on how to prevent this, see the Simplify values (no references) CIF to CIF transformation.

Optimality

Not all simplifications that could potentially be performed are implemented in this transformation.

Annotations

This transformation does not process or add any annotations. The values of annotation arguments may be simplified. Invariants that don't add any restrictions may be removed, also removing their annotations.

Simplify values (optimized)

This CIF to CIF transformation simplifies CIF specifications, by applying same value-related simplifications as the Simplify values CIF to CIF transformation, but with optimizations for literal expressions.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

One of the simplifications that the CIF to CIF transformation to Simplify values performs, is the evaluation of constant (sub-)expressions. For large literal expressions, such as arrays with thousands or even millions of elements, this is expensive, without having any effect. For instance, consider list/array literal `[1, 2, 3]`. Evaluating the list/array and the integer elements, and then reconstructing the literal list/array expression and literal integer expressions, has no effect. The result is the same list/array literal as the original.

This transformation detects literal expressions, and doesn't further simplify them, thus greatly improving performance for large literal expressions. Literal expressions are detected using a static analysis. Due to using static analysis, some literals may not be simplified, while they would have been simplified if this literal detection optimization was not used. For instance, a set literal `{1, 2, 1}` is simplified to `{1, 2}` without optimizations, and is not simplified with optimizations. In this case, static analysis does not account for duplicate elements of the set.

The following differences can be observed for literal optimization, with respect to not using literal optimization:

- Negations of integer literals are not normalized, although this only affects the internal representation of negative integer values, and may not be externally visible to end users.
- Real literals are not normalized. For instance, `1e3` is not normalized to `1000.0`.
- For set literals, duplicate elements are not removed.
- For dictionary literals, duplicate keys are not removed, and the dictionary literal is not further

simplified. Without literal optimization, duplicate keys would lead to an error during simplification.

Renaming

n/a

Size considerations

This transformation tries to simplify the specification, possibly reducing its size.

All uses of constants lead to constant (sub-)expressions, and they are evaluated. This leads to constants being inlined. For constants with large literal values, this may significantly increase the size of the specification, especially if the constant is used more than once. For information on how to prevent this, see the Simplify values (no references, optimized) CIF to CIF transformation.

Optimality

Not all simplifications that could potentially be performed are implemented in this transformation.

The optimizations for literals that are performed influence the optimality of the simplification, as described above.

Annotations

This transformation does not process, add, or remove any annotations. The values of annotation arguments may be simplified.

Simplify values (no references)

This CIF to CIF transformation simplifies CIF specifications, by applying same value-related simplifications as the Simplify values CIF to CIF transformation, but without simplifying reference expressions.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given

order):

- Eliminate component definition/instantiation

Implementation details

The CIF to CIF transformation to Simplify values, transforms the following specification:

```
const int x = 3;
invariant x - 5 + 1 > 0;
```

into the following simplified specification:

```
const int x = 3;
invariant false;
```

This transformation however, does not take the values of declarations into account, and thus for instance does not fill in the values of constants, algebraic variables, etc. Therefore, this transformation transforms that same specification into the following simplified specification:

```
const int x = 3;
invariant x - 4 > 0;
```

This variant can be useful, as it prevents the use of constants from being eliminated. This makes it possible to change the value of the constant in one place (the one place being the declaration of the constant), and have all uses of that constant automatically change with it (as they still refer to the constant). If the value of the constant would be filled in as well, changing the value of the constant would be more work, as all those places where it was filled in, need to be replaced as well. In the above example, the issue is even bigger, as the result is simplified even further, eliminating the entire comparison to a single boolean value.

Another issue related to simplification of constants, is that inlining them can be very expensive. If a constant has a large literal as its value, the large literal is inlined wherever the constant is used. This may lead to significantly larger specifications. Also, tools that use the resulting model may handle constants in a special way to ensure they are evaluated only once, but may not do so for large literals that occur in the middle of other expressions. As such, simplification/inlining of constants may significantly reduce performance.

The reference expressions that are not simplified (filled in) include among others the following:

- Values of constants.
- Values of discrete variables.
- Values of algebraic variables.
- Values of continuous variables.

- Values of input variables.
- Derivatives of continuous variables.
- Values of local variables of functions.
- Values of parameters of functions.
- User-defined functions.

Renaming

n/a

Size considerations

This transformation tries to simplify the specification, possibly reducing its size.

Optimality

Not all simplifications that could potentially be performed are implemented in this transformation.

Annotations

This transformation does not process, add, or remove any annotations. The values of annotation arguments may be simplified.

Simplify values (no references, optimized)

This CIF to CIF transformation simplifies CIF specifications, by applying same value-related simplifications as the Simplify values CIF to CIF transformation, but without simplifying reference expressions, and with optimizations for literal expressions.

Supported specifications

This transformation supports a subset of CIF specifications. The following restrictions apply:

- Component definitions and component instantiations are not supported.

Preprocessing

No preprocessing is currently performed by this CIF to CIF transformation. To increase the subset of specifications that can be transformed, apply the following CIF to CIF transformations (in the given order):

- Eliminate component definition/instantiation

Implementation details

This CIF to CIF transformation simplifies CIF specifications, by applying same value-related simplifications as the Simplify values CIF to CIF transformation.

However, it does not simplifying reference expressions. For details, see the Simplify values (no references) CIF to CIF transformation. Furthermore, it simplifies with optimizations for literal expressions. For details, see the Simplify values (optimized) CIF to CIF transformation.

Renaming

n/a

Size considerations

This transformation tries to simplify the specification, possibly reducing its size.

Optimality

Not all simplifications that could potentially be performed are implemented in this transformation.

The optimizations for literals that are performed influence the optimality of the simplification, as described above.

Annotations

This transformation does not process, add, or remove any annotations. The values of annotation arguments may be simplified.

4.6.2. CIF merger

The CIF merger can be used to merge two or more CIF specifications into a single CIF specification. The result of the merger is essentially the parallel composition of the input specifications. Unlike the CIF explorer and Event-based synchronous product tools however, the parallel composition is not unfolded into a state space.

It can for instance be useful to merge a synthesized supervisor with a timed or hybrid plant, to validate the supervisor against that more detailed plant, using interactive simulation and visualization on the merged specification.

The specifications that are being merged can share events and variables, making it possible to have interaction between the merged specifications.

Starting the transformation

The merger can be started in the following ways:

- Use the Eclipse IDE to merge multiple files:
  - Select a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab by left clicking on it.
  - Select additional `.cif` and/or `.cifx` files by left clicking on them while pressing the `Ctrl` or `Control` key.
  - Make sure at least two `.cif` and/or `.cifx` files are selected.
  - Right click one of the `.cif` or `.cifx` files and choose **Merge CIF specifications...**.
- Use the Eclipse IDE to merge a single file with itself (allowed, but practically not that useful):
  - Right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF miscellaneous tools › Merge CIF specifications...**.
  - Right click an open text editor for a `.cif` or `.cifx` file and choose **CIF miscellaneous tools › Merge CIF specifications...**.
- Use the `cifmerge` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifmerge` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input files*: The absolute or relative local file system path to the input CIF specifications. These are the files to merge.
- *Output file*: The absolute or relative local file system path to the merged/output CIF specification. If not specified, the output file defaults to `merged.cif`. If the merger is executed via the Eclipse GUI, the output file, if specified as a relative path (or just a file name), is resolved relative to the directory that contains the first input file.
- *Merged specification is partial*: Specifies whether to make the merged specification a partial specification. By default, the merged specification is a non-partial specification. The option can be enabled to make the merged specification a partial specification.
- *Remove controller properties annotations*: Enable this option to remove the controller properties annotations of the input specifications, so that the merged specification has no such annotations. Disable the option to keep the controller properties annotations of the input specifications and merge them into the merged specification. By default, the option is enabled, as there is no guarantee that the controller properties annotations will apply to the merged specification.



Disable this option with care!

If you are sure the controller properties annotations still apply after the merge, you can disable the option to preserve the annotations. Note that if the option is disabled, and multiple input specifications have a controller properties annotation, then the merged specification will be invalid (see the merge compatibility section for more information).

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be transformed:

- Eliminate component definition/instantiation
- Push SVG file declarations into other CIF/SVG declarations
- Push print file declarations into print declarations

If only one input file is provided to the CIF merger, essentially that single input file is preprocessed and written as output, without any merging taking place.

Example

Consider the following supervisor:

```
group button:
  uncontrollable u_pushed, u_released;
end

group lamp:
  controllable c_on, c_off;
end

group timer:
  controllable c_start;
  uncontrollable u_timeout;
end

supervisor automaton timed_lamp:
  location s0:
    initial;
    edge button.u_pushed goto s1;
    edge button.u_released;

  location s1:
    edge lamp.c_on goto s2;
    edge button.u_pushed, button.u_released;

  location s2:
    edge timer.c_start goto s3;
```

```
    edge button.u_pushed, button.u_released;

  location s3:
    edge timer.u_timeout goto s4;
    edge button.u_pushed, button.u_released;

  location s4:
    edge lamp.c_off goto s0;
    edge button.u_pushed, button.u_released;
end
```

and the following timed plant:

```
plant automaton button:
  uncontrollable u_pushed, u_released;

  location released:
    initial;
    edge u_pushed goto pushed;

  location pushed:
    edge u_released goto released;
end

plant automaton lamp:
  controllable c_on, c_off;

  location off:
    initial;
    edge c_on goto on;

  location on:
    edge c_off goto off;
end

plant automaton timer:
  controllable c_start;
  uncontrollable u_timeout;

  cont t der 1.0;

  location idle:
    initial;
    edge c_start do t := 0.0 goto running;

  location running:
    edge u_timeout when t >= 2.0 goto idle;
end
```

Merging these two specifications results in the following merged specification:

```
plant automaton button:
  uncontrollable u_pushed;
  uncontrollable u_released;
  location released:
    initial;
    edge u_pushed goto pushed;
  location pushed:
    edge u_released goto released;
end
plant automaton lamp:
  controllable c_on;
  controllable c_off;
  location off:
    initial;
    edge c_on goto on;
  location on:
    edge c_off goto off;
end
plant automaton timer:
  controllable c_start;
  uncontrollable u_timeout;
  cont t der 1.0;
  location idle:
    initial;
    edge c_start do t := 0.0 goto running;
  location running:
    edge u_timeout when t >= 2.0 goto idle;
end
supervisor automaton timed_lamp:
  location s0:
    initial;
    edge button.u_pushed goto s1;
    edge button.u_released;
  location s1:
    edge lamp.c_on goto s2;
    edge button.u_pushed, button.u_released;
  location s2:
    edge timer.c_start goto s3;
    edge button.u_pushed, button.u_released;
  location s3:
    edge timer.u_timeout goto s4;
    edge button.u_pushed, button.u_released;
  location s4:
    edge lamp.c_off goto s0;
    edge button.u_pushed, button.u_released;
end
```

The supervisor specification contains three *skeletons*: groups that only contain event declarations,

and have no behavior. The skeletons represent the plants of the system, which are controlled by the supervisor. The supervisor specification also contains a supervisor automaton, which waits for the button to be pushed, then turns on the lamp and start the timer, waits for the timer to time out, turns the lamp back off, and keeps repeating this behavior. Pushing or releasing the button during the cycle has no effect.

The timed specification contains the same plants, with the same events, but here the plants are actual automata with behavior, instead of just skeleton groups. For the button and lamp, the events occur in alternating sequences. The timer has a clock that ensures that the timeout happens two time units after the timer is started.

Since both specifications contain the same events (based on their absolute names), these events are merged together. This ensures that uses of those events become linked (or coupled), and all refer to the single merged event in the merged specification. This means that the events used in the supervisor are the same events as used the timed plant. This also means that they synchronize, and the supervisor thus controls the timed plants.

Shared events and variables

The example above shows how shared events can be used to link (or couple) multiple specifications through merging.

Similarly to the way events can be merged, it is possible to merge input variables with other variables, as long as the types are compatible. That is, an input variable can be merged with another input variable, a discrete variable, a continuous variable, an algebraic variable, a constant, or a location. The input variable is then a sort of *skeleton* definition that is merged with an actual definition (for instance an algebraic variable), which provides the value(s).

There are thus two ways for merged specifications to become linked (or coupled): by means of shared events and by means of shared variables (including constants).

For further details and restrictions, see the Merge compatibility section.

Merge compatibility

In general, objects (components, declarations, etc) that only occur in one of the specifications being merged, are simply copied to the merged specification. If in two (or more) specifications objects with the same absolute name are present, they need to be merged. Not all objects can be merged with all other objects, i.e. not all objects are *merge compatible*. The following table gives an overview of what can be merged:

Merge with	Group	Automaton	Event	Input var	Discrete var	Continuous var	Algebraic var	Constant	Location	Type declaration	Enumeration	Enumeration literal
Group	yes	yes	no	no	no	no	no	no	no	no	no	no
Automaton	yes	no	no	no	no	no	no	no	no	no	no	no
Event	no	no	yes	no	no	no	no	no	no	no	no	no
Input var	no	no	no	yes	yes	yes	yes	yes	yes	no	no	no
Discrete var	no	no	no	yes	no	no	no	no	no	no	no	no
Continuous var	no	no	no	yes	no	no	no	no	no	no	no	no
Algebraic var	no	no	no	yes	no	no	no	no	no	no	no	no
Constant	no	no	no	yes	no	no	no	yes	no	no	no	no
Location	no	no	no	yes	no	no	no	no	no	no	no	no
Type declaration	no	no	no	no	no	no	no	no	no	yes	yes	no
Enumeration	no	no	no	no	no	no	no	no	no	yes	yes	no
Enumeration literal	no	no	no	no	no	no	no	no	no	no	no	yes

For components (i.e. groups and automata) that are present in more than one specification, the contents are recursively merged into a single component. The contents of components consists not only of their declarations (such as events) and their sub-components (for groups), but also of their invariants, initialization predicates, marker predicates, equations, locations (for automata), etc.

As the button/lamp example above shows, automata can be merged with groups (often acting as skeletons), resulting in an automaton with the contents of the group merged into it. Since automata

can’t contain sub-components and user-defined functions, groups with sub-components and/or user-defined functions can not be merged with automata.

Groups can also be merged with other groups. Automata however can not be merged with other automata, as that may lead to the merge of conflicting behavioral specifications.

Events can be merged with other events. Each event must then either be a channel in all of the specifications in which it occurs, or in none of them. If it is a channel, the data type must be exactly the same in all specifications in which it is declared. Events that have different controllability in different specifications, can also not be merged.

Constants can be merged with other constants. However, the constants must then have the exact same type and value in all the specifications in which they occur. This restriction prevents the merge of conflicting values. Due to this restriction, constants that have a type of which the values can not be compared for equality (e.g. function types), are not supported (can not be merged).

As mentioned in the Shared events and variables section, an input variable can be merged with another input variable, discrete variable, continuous variable, algebraic variable, constant, or location. Input variables can only be merged with other objects that have the exact same type.

A location can be merged with an input variable that has a boolean type. The input variable is then a placeholder for the location, indicating whether the location is the current location of its automaton.

It is not allowed to merge two algebraic variables, two discrete variables, or two continuous variables, as that may lead to the merge of conflicting values.

In general, only one of the specifications defines the variable as a concrete variable that defines a value (e.g. as a discrete, continuous, or algebraic variable). All the other specifications that have that same variable must declare it as an input variable (essentially a *placeholder*). It is however allowed to merge two or more constants with the same type and the same value.

Type declarations can be merged with other type declarations, as long as they have the exact same type.

Enumerations can be merged with other enumerations, as long as they are compatible. Two enumerations are compatible if they have the same number of literals, with the exact same names, in the same order.

Enumeration literals can only be merged as the result of the enumerations of which they are a part being merged. Merging two literals from different enumerations (which then have different names) is not supported.

Type declarations and enumerations can be merged, as long as the type of the type declaration is compatible with the enumeration.

Invariants can never be merged with other objects of the same name.

User-defined functions can never be merged with other objects of the same name.

Specifications are treated as groups.

By default, the merged specification never has a controller properties annotation, even if an input specification has one. If the *Remove controller properties annotations* option is disabled, the controller properties annotations of the input specifications are preserved and merged into the merged specification. Make sure to only use this option if you are sure that the information captured in such annotations is preserved in the merged specification! If the option is disabled, and multiple input specifications have a controller properties annotation, then the merged specification will have multiple such annotations. This is not allowed and thus the merge will fail. If you disable the option, make sure only one of the input specifications has a controller properties annotation.

By default, the merged specification is a non-partial specification. The *Merged specification is partial* option can be enabled to make the merged specification a partial specification.

Merge problems

Even when two or more CIF specifications are merge compatible, as described above, the resulting merged CIF specification can still be invalid. In such cases, merging fails. For instance, consider the following two CIF specifications:

```
input int x;
alg int y = x;
```

```
alg int x = y;
input int y;
```

Merging them would result in:

```
alg int x = y;
alg int y = x;
```

But this merged specification is invalid, as x is defined in terms of y, which is defined in terms of x, leading to a loop (definition/use cycle).

A second example of merge problems is two CIF specifications that have SVG output mappings for the same SVG element id and attribute, for the same SVG image file. In the individual CIF specifications there are no duplicate output mappings, but in the merged specification there are.

A third example of merge problems is a specification that has an SVG input mapping for an input variable that is merged with a non-input variable. In the merged specification, the SVG input mapping updates a non-input variable, which is not allowed.

The merged specification is also checked for any problems with annotations, as annotations may be on different objects after merging, and not all annotations are supported for all objects. An example of this is merging multiple specifications with a controller properties annotation, provided the option to remove such annotations is disabled; see for more information the section on merge compatibility.

4.6.3. CIF event disabler

The CIF event disabler tool can be used to disable certain events of a CIF specification, or even to disable events not currently present in a CIF specification. The tool has various options that can be used to control the working of the tool, and influence its output.

Usage scenario

Assume a system with some sensors and actuators. From the view of the supervisor (or controller in general), the sensors are uncontrollable, and the actuators are controllable. Now assume a controller is being designed for this system, and it is to be merged with a simulation model of the uncontrolled system, that is being designed as well.

The situation may arise that the controller only controls a part of the uncontrolled system. This may for instance be the case if the designer of the controller starts with a small controller and incrementally extends it to cover more and more functionality of the system. The designer may have already made a full simulation model of the uncontrolled system beforehand. Alternatively, if one designer develops the controller and another develops the simulation model of the uncontrolled system, the simulation model may be finished well before the controller for the entire system is finished.

A problem arises when a controller that controls only part of the system, is merged with a simulation model of the entire uncontrolled system. In the uncontrolled system, the actuators can usually be turned on and off arbitrarily, as they are not yet controlled in any way. If the controller doesn’t control certain actuators, the behavior of those actuators is not restricted, and during simulation the events to turn the actuators on and off can happen at any time. Events are urgent, that is, they take precedence over passage of time, and thus happen 'immediately'. Events that are always enabled thus prevent passage of time. The simulation will turn the actuators on, off, on again, off again, etc.

To solve this problem, the controllable events of the uncontrolled system that are not restricted in any way by the controller need to be disabled. The CIF event disabler tool makes this possible.

This is however not the only usage scenario. The same controller may be used for implementation. For the implementation, a mapping from events to variables that represent the I/O ports may be needed. This mapping could be a CIF specification, with an automaton that performs the mapping. Similar to having a complete simulation model, a complete mapping specification for the implementation may be available. If in such mappings the actuator events are always enabled, this may lead to similar problems as for the simulation, if the controller only controls part of the system. Here too, the event disabler can be used to restrict such actuator events from causing trouble.

Starting the tool

The tool can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF miscellaneous tools › Disable events of CIF specification...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF miscellaneous tools › Disable events of CIF specification...**.
- Use the `cifevtdis` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifevtdis` command line tool.

Supported specifications

The event supports all CIF specifications.

Preprocessing

The following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be modified:

- Eliminate component definition/instantiation

Options

Besides the general application options, this application has the following options:

- *Input file*: The absolute or relative local file system path to the input CIF specification. If we relate this to the above usage scenario, the input file should be the controller that only controls part of the system.
- *Output file*: The absolute or relative local file system path to the output CIF specification. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.disabled.cif` file extension is added.
- *Event names*: The absolute names of the events. How these supplied events are used depends on the *Event usage* option. Multiple events may be specified, separated by commas and/or spaces.
- *Event names file*: The absolute or relative local file system path of a file with the absolute names of the events. How these supplied events are used depends on the *Event usage* option. Multiple events may be specified in the file, on separate lines. Empty lines are ignored. Lines starting with a `#` character are ignored as well, and can thus be used to add comments.
- *SVG input events*: Enable this option to use the SVG input events. How the events are used depends on the *Event usage* option.
- *Event usage*: This option can be used to specify how the supplied events are to be used. That is, it specifies which events to disable. See the Specifying the events to disable section for more information.

- *Include input specification*: Should the output file contain only the new automaton that is created to disable the events, or should it include the input specification as well? See the Output section for more information.

At least one of the *Event names*, *Event names file*, or *SVG input events* options must be used to supply events to the tool. It is allowed to use multiple of these options, to combine events from multiple sources, and supply all of them to the tool.

Specifying the events to disable

The *Event names*, *Event names file*, and *SVG input events* options can be used to supply events to the tool. How these events are used (or interpreted) depends on the *Event usage* option. Several alternatives are available:

- Disable the supplied events, regardless of the alphabet of the input specification (`disable`)
- Disable all supplied events, that are not in the alphabet of the input specification (`alphabet`)

The texts between parentheses at the end of the alternatives indicate the command line option values that correspond with the alternatives. The default is `alphabet`.

The most straightforward alternative is to manually specify the events to `disable` (the first alternative in the above list). The supplied events are then disabled.

Manually specifying the events to disable can be cumbersome. Especially if the controller is developed incrementally, this can be a burden. The more events the controller does restrict, the less events should be disabled using the event disabler. In this case it may be easier to list the controllable events of the uncontrolled system that the controller may or may not restrict. That is, the `alphabet` (the second alternative in the above list) of controllable events of the uncontrolled system is specified. Supplied events that are not in the alphabet of any of the automata of the input specification, are disabled.

The following table summarizes which events get disabled by the tool:

Event usage	Event supplied	Event in alphabet	Event is disabled by tool
disable	yes	yes/no	yes
disable	no	yes/no	no
alphabet	yes	yes	no
alphabet	yes	no	yes
alphabet	no	yes/no	no

Output

If the *Include input specification* option is enabled, the output specification is a copy of the input

specification, which is then modified to disable certain events. For events that are disabled by the tool, but did not exist in the input specification, an event declaration is added. If the option is disabled, the output specification starts empty. For all events are disabled, an event declaration is added. Event declarations are added in their proper scopes, based on their absolute names, by adding groups around them as needed. The added event declarations and groups do not have any annotations.

By default, the *Include input specification* option is disabled. This ensures that the output specification is small. The typical scenario is to merge the input specification, event disabler output specification, and simulation model together using the CIF merger tool. Disabling the *Include input specification* specification option then gives the best performance for the tool chain.

If any events are disabled by the tool, a new automaton is created, and this automaton is added to the output specification. The automaton is named `event_disabler`, but may be renamed if it conflicts with an already existing object with the same name. The automaton has no supervisory kind, and has the disabled events as alphabet. It has a single nameless location, with a single self loop for all of the events that are disabled. The guard of the self loop is `false`, which results in the events being globally disabled. The newly added automaton and all its elements do not have any annotations.

If the *Include input specification* option is enabled, and events are disabled that did not exist in the input specification, event declarations are added to the output specification for those events. If the *Include input specification* option is disabled, event declarations are added for all events that are disabled. Event declarations that are added and existed in the input specification (in case the *Include input specification* option is disabled), inherit their controllability from the input specification. All other events that are added are declared as controllable events if their name starts with `c_`, are declared as uncontrollable events if their name starts with `u_`, and are declared as neither controllable nor uncontrollable otherwise.

The event disabler does not process or add any annotations. The output specification never has a controller properties annotation, even if the input specification has one and the input specification is included in the output.

4.6.4. CIF explorer

The CIF explorer unfolds the state space expressed by a CIF specification, in an untimed setting. It thereby computes the untimed state space, consisting of all reachable states and the transitions between them.

It takes a CIF model and iteratively explores the states using event transitions, unfolding the state space. Time transitions are never taken. The explorer will return a deadlock state when forced into a time step. Continuous variables are allowed, but behave much like discrete variables due to lack of time steps.

Available output forms include a CIF model with an automaton representing the state space, and a report with details of the found states and transitions.

The statespace is computed in terms of the states of the CIF specification, based on the current locations of its automata and the values of its variables. The resulting statespace automaton is an NFA or a DFA, but is not necessarily minimal. You can minimize it using the DFA minimizer,

assuming the state space satisfies that tool's preconditions.

Starting the program

The explorer can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF miscellaneous tools › Explore untimed state space...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF miscellaneous tools › Explore untimed state space...**.
- Use the `cifexplorer` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cifexplorer` command line tool.

Options

Besides the general application options, the application has the following options:

- *Input file path*: The absolute or relative file system path to the input CIF specification.
- *Remove duplicate transitions*: Removes duplicate transitions from the state space, before printing statistics and writing output files. Two transitions are duplicates if they have the same source and target states, and the same event. The communication value, if any, is ignored. Enabling this option removes duplicate transitions from the state space, possibly reducing the state space. But, enabling this option does not guarantee that the resulting state space automaton is minimal. This option is enabled by default.
- *Enable statistics*: Enable printing of statistics of the resulting state space to the console. Statistics include the number of states and the number of transitions of the resulting state space. It is recommended to enable the option to remove duplicate transitions when printing statistics. This option is enabled by default.
- *Enable CIF output*: Enable output as a CIF automaton, with states represented by locations and transitions by edges. This option is enabled by default. See for more information the section on output as a CIF automaton.
- *Add state annotations*: Add state annotations to the locations of the CIF automaton in the output CIF specification. This option is enabled by default. See for more information the section on output as a CIF automaton.
- *Output file path*: The absolute or relative file system path of the output CIF specification with the generated state space as a CIF automaton.
- *Name*: The name of the resulting statespace automaton. If not specified, it defaults to `statespace`. If the resulting statespace automaton has a name that conflicts with an existing declaration, it is automatically renamed to have a non-conflicting name. See for more information the section on

output as a CIF automaton.

- *Enable report*: Enable writing a detailed report on the found states. This option is disabled by default.
- *Report file path*: The absolute or relative file system path of the report file. If specified, option *Enable report* is implied.
- *Print progress*: The number of states to process before printing progress information. Must be a non-negative integer number. May be `off` to disable progress information. The default is to print progress information after processing 1000 states.

The CIF output file is written if the *Enable CIF output* option is set, or if a path is supplied with the *Output file path* option. If the latter is given, its value is used as the path for writing the CIF file. If only the *Enable CIF output* option is set, the value of the *Input file path* option is used, where the `.cif` extension is removed (if present), and a `_statespace.cif` suffix is added.

The report file is written if the *Enable report* option is set, or if a path is supplied with the *Report file path* option. If the latter is given, its value is used as the path for writing the report file. If only the *Enable report* option is set, the value of the *Input file path* option is used, where the `.cif` extension is removed (if present), and a `_report.txt` suffix is added.

If both the CIF output and the report file output are disabled, the resulting state space is not outputted at all. In such cases, the state space explorer acts as a verifier for runtime errors, which can make the exploration fail. If exploration completes without errors, the explorer has verified that no runtime errors occur. Runtime errors include failures to compute values, such as division by zero, as well as assignments that assign values to variables that are outside the allowed bounds/ranges of the variables.

Supported specifications

The explorer supports a subset of CIF specifications. The following restrictions apply:

- Usage of distribution types and distribution standard library functions is not supported.
- Usage of derivatives is not supported.
- External user-defined functions are not supported.
- Input variables are not supported.
- Specifications with more than 2^{31} - 1 = 2,147,483,647 potential initial states are not supported.

The following information from the specification is ignored:

- Automaton and invariant supervisory kinds. A warning is printed if a requirement automaton or requirement invariant is encountered.
- Controllability of events.
- I/O declarations are ignored. A warning is printed if a CIF/SVG input declaration is encountered.
- Annotations are ignored. If outputting a CIF model is enabled, the produced model may have annotations. See for more information the section on output as a CIF automaton.

Preprocessing

Prior to exploration, the following CIF to CIF transformations are applied as preprocessing (in the given order), to increase the subset of CIF specifications that can be explored:

- Remove I/O declarations
- Remove annotations
- Eliminate component definition/instantiation
- Eliminate automaton `self` references

In addition it applies the Simplify values (no references, optimized) CIF to CIF transformation to speed up processing.

Output as a CIF automaton

If outputting a CIF model is enabled, a CIF specification is created and written to a file. The state space is represented by a state space automaton, each state is represented by a location of that state space automaton, and the transitions between states are represented as edges between locations. The state space automaton is named as configured using the *Name* option. If the state space is empty, the state space automaton has a single non-initial location.

If adding state annotations is enabled each location is annotated with a state annotation. The state annotation indicates the state of the input model represented by the location of the state space automaton. That is, each state annotation indicates the current location of each automaton from the input specification, and the current value of each variable from the input specification. If the state space is empty, the single location in the state space automaton does not have a state annotation, regardless of whether the option to add state annotations is enabled or not. The resulting CIF specification does not have any other annotations on any of its elements.

4.6.5. CIF to yEd transformer

The CIF to yEd transformer can be used to transform CIF specifications to yEd diagrams. Several kinds of diagrams can be generated. These diagrams can be used to better understand the specification, to communicate the specification to other, or to be included in reports.

The yEd Graph Editor is an application that can be used to view, create, edit, arrange, import and export various kinds of diagrams. It is freely available and runs on Windows, Linux, and macOS.

The yEd diagrams are generated as GraphML files (`*.graphml` files), using yEd specific extensions to specify the graphical representation.

Starting the transformation

The transformation can be started in the following ways:

- In Eclipse, right click a `.cif` or `.cifx` file in the *Project Explorer* tab or *Package Explorer* tab and choose **CIF miscellaneous tools › Convert CIF to yEd diagram...**.
- In Eclipse, right click an open text editor for a `.cif` or `.cifx` file and choose **CIF miscellaneous tools › Convert CIF to yEd diagram...**.
- Use the `cif2yed` tool in a ToolDef script. See the scripting documentation and tools overview page for details.
- Use the `cif2yed` command line tool.

Options

Besides the general application options, this application has the following options:

- *Input file path*: The absolute or relative local file system path to the input CIF specification.
- *Output file path*: The absolute or relative local file system path to the output yEd/GraphML diagram file. If not specified, defaults to the input file path, where the `.cif` file extension is removed (if present), and a `.kind.graphml` file extension is added. For the added default file extension, `kind` is replaced by the diagram kind (see other option), e.g. `model`, resulting in file extension `.model.graphml`. If multiple diagrams are generated, the output file path may not be specified. In such cases, either use the implicit/default file names, or generate one diagram at a time, allowing you to specify an explicit output file path for each of them.
- *Diagram kinds*: The kinds of diagrams to generate. The default is to generate all the different kinds of diagrams. See the Diagram kinds section for details.
- *Syntax highlighting*: Whether to apply syntax highlighting (if option is enabled), or use plain text (if option is disabled). By default, syntax highlighting is enabled. For large diagrams, disabling syntax highlighting can significantly improve the performance, when displaying the diagrams in yEd.
- *Transparent edge labels*: Enable option to use a transparent background for edge labels. Disable option to use a light gray background. By default, the option is disabled (a light gray background is used). See the Layout section below for further details.
- *Model filters*: The kinds of features to include in the diagram. Only applies to model diagrams. See the Model diagram section for details.
- *Relations*: The kinds of relations to include in relations diagrams. The default is to include all kinds of relations. See the Relations diagram section for details.

Supported specifications

For model diagrams:

- This transformation supports all CIF specifications.

For relations diagrams:

- CIF specifications with component parameters are not supported.

The transformation does not process any specific annotations. However, annotations may be included in some diagrams, such as model diagrams.

Preprocessing

No preprocessing is currently performed by this transformation.

Diagram kinds

Several types of diagrams can be created. They are each described below. Using the *Diagram kinds* option (see the Options section above), generation of certain kind of diagrams can be enabled or disabled.

In the option dialog, each diagram kind can be enabled and disabled individually, using a checkbox.

From the command line, using the `-k` or `--diagram-kinds` option, the names of the diagram kinds, as indicates for each diagram kind in the sections below, should be used, separated by commas. By default, i.e. when the `-k` or `--diagram-kinds` option is not used, diagrams are generated for all the different kinds. As an example, consider `-k model` or `--diagram-kinds=model`. Only a model diagram is generated, while all other diagrams are not generated.

The diagram kinds that are specified using the option replace the default diagram kinds. However, it is also possible to specify additions and removals relative to the default features, by prefixing features with a `+` or `-` respectively. For instance, when using `-k -model` or `--diagram-kinds=-model` the default diagrams are generated, except for the model diagram (due to `-model`). Replacements (no prefix) may not be combined with additions/removals (`+` or `-` prefix).

Specifying a replacement diagram kind twice leads to a warning being printed to the console. Adding a diagram kind that is already present or removing a diagram kind that is not present, also leads to a warning being printed. A warning is also printed to the console if all diagram kinds are disabled, and thus no output is generated.

Model diagram

The option value to use on the command line, for this kind of diagrams, is `model`.

A diagram of the model itself, with all its details. That is, the diagram can be used instead of the textual representation of the model, without loss of details. Changes with respect to the textual representation include among others elimination of imports, and the loss of comments and textual layout.

All components (automata, groups, automaton instantiations, and group instantiations) and components definitions (automata and group definitions) are visualized in a hierarchical manner, as rectangles contained in other rectangles. Automaton and group instantiations are grouped together into a single node in the diagram. All of them can be closed (collapsed) and opened

(expanded) in yEd, allowing to hide unimportant details while focusing on only a part of the diagram. The header of the component or component definition is shown at the top of the rectangle, while for instantiations the word `Instantiations` is used as header. For groups and group definitions an italic font is used, while for automata, automaton definitions, and automaton/group instantiations, a regular font is used. The background (of the header) is colored as follows:

- A component (group/automaton)
- A component definition (group/automaton definition)
- A component instantiation (group/automaton instantiation)

For instantiations, the complete instantiations are shown as code block in the body of the rectangle.

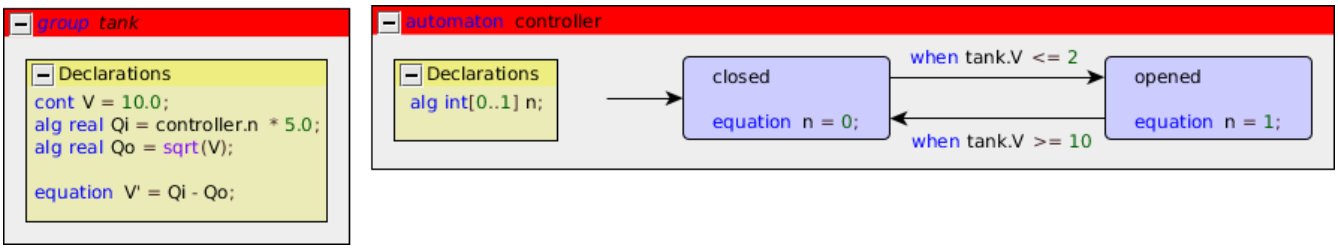
For the declarations (e.g. variables), invariants, I/O declarations (e.g. SVG declarations), etc a block of code is used. Code blocks use header and background colors that are distinct from the colors used for components, component definitions, and component instantiations:

- A code block (header)
- A code block (background)

For automata, the locations are visualized as rectangles with rounded colors. Their names (if applicable), invariants, equations, etc are shown as code in the rectangles. The initialization of the location (if present and not `false`) is visualized as an incoming arrow without source location, with a label indicating the initialization predicates if they are not `true`. Edges are visualized as arrows between locations, with arrows to the source location itself for self loop edges. The events, guards, `now` keywords (for urgent edges), and updates are put as labels on the edges. Locations use a background color that is distinct from the colors used for components, component definitions, and code blocks:

- A location (background)

As an example consider the following diagram of the CIF `tank` example:



Using the *Model filters* option (see the Options section above), certain features of the model can be included or excluded from the diagram. The table below indicates the features that can be enabled and disabled using the option. The first column lists the names to use on the command line. The second column indicates whether the feature is enabled (included) by default. The third column describes what parts of the model are included/excluded when the feature is enabled/disabled.

Name	Default	Description
`io`	no	The I/O declarations, e.g. SVG declarations.

Name	Default	Description
`decls`	yes	The declarations of the components. Includes the alphabet (if explicitly specified) and monitors of the automata, event declarations, constant declarations, variable declarations, function declarations, invariants, initialization predicates, marker predicates, equations, etc.
`loc-decls`	yes	The declarations of the locations. Includes the invariants, marker predicates, equations, and urgency.
`guards`	yes	The guards of the edges. Includes also the urgency of the edge (the `now`) keyword.
`updates`	yes	The updates of the edges.

In the option dialog, each of the features can be enabled and disabled individually, using a checkbox.

From the command line, using the `-f` or `--model-filters` option, the names of the different features, as indicates in the first column of the table above, should be used, separated by commas. The features indicated as default in the second column of the table, are enabled when default settings are used, i.e. when the `-f` or `--model-filters` option is not used. As an example, consider `-f guards,updates` or `--model-filters=guards,updates`. Only the guards and updates are included, while all the other features are excluded.

The features that are specified using the option replace the default features. However, it is also possible to specify additions and removals relative to the default features, by prefixing features with a `+` or `-` respectively. For instance, when using `-f -loc-decls,+io` or `--model-filters=-loc-decls,+io` the default features are used, with the declarations of the locations disabled (due to `-loc-decls`), and with I/O declarations enabled (due to `+io`). Replacements (no prefix) may not be combined with additions/removals (`+` or `-` prefix).

Specifying a replacement feature type twice leads to a warning being printed to the console. Adding a feature that is already present or removing a feature that is not present, also leads to a warning being printed.

Relations diagram

The option value to use on the command line, for this kind of diagrams, is `relations`.

A diagram of only the event and data relations. The diagram can be used to see which events are

used by which automata (are in the alphabet of those automata). It can also be used to see which data (variables, locations, etc) exist in the model and where they are used. Furthermore, it can also be used to see how events and data are passed via automaton and group instantiations. Events, variables, and locations can thereby potentially be renamed if the parameters of the corresponding automaton or group definitions have different names than the events, variables, and locations that are passed via the instantiations.

All components (automata, groups, automaton instantiations, and group instantiations) and component definitions (automaton and group definitions) are visualized in a hierarchical manner, as rectangles contained in other rectangles. Automaton and group instantiations are unfolded to their contents (the body of the corresponding automaton or group definition). Essentially, it is as if component definition/instantiation has been eliminated, but the parameters are kept. Within components and component definition bodies, the components (including the component instantiations) and declarations (event declarations, event parameters, algebraic variables, algebraic parameters, locations, input variables, input parameters, etc) are separated from the component definitions, using additional wrapping boxes. Such wrapping boxes are only added if both components and component definitions are present, and thus separation is needed.

However, all internal details of component instantiations are omitted, as they can be seen in the graphical representations of component definitions (that are instantiated by the instantiations). That is, only the external connections are included, as they show the actual relations (definition here and use there, synchronizations between components, etc). By excluding the internal details, duplication is avoided for multiple instantiations of the same definition. All empty components and components definitions are also excluded, as they only clutter the diagram, and have no relations whatsoever.

All components can be closed (collapsed) and opened (expanded) in yEd, allowing to hide unimportant details while focusing on only a part of the diagram. By default, all components are closed (collapsed). This allows seeing the top level structure. By selectively opening (expanding) components, more details can be shown. It is also possible to focus on a single component, to show only that component and its contents, by selecting **Grouping › View Group Content** in yEd, while a closed or opened component is selected.

The header of the component is shown at the top of the rectangle. It includes the name of the component or component definition, whether it is a group or automaton, whether it is a definition or not, etc. The parameters of component definitions, and the arguments of component instantiations, are all omitted from the headers. For wrappers either *Components* or *Component definitions* is used as header text. For groups, group definitions, and group instantiations an italic font is used, while for automata, automaton definitions, automaton instantiations, and wrapping boxes, a regular font is used. The background of the header is colored as follows:

- A component (group/automaton)
- A component definition (group/automaton definition)
- A component instantiation (group/automaton instantiation)
- A wrapping box for components (incl. instantiations) or component definitions

Event declarations, discrete variables, continuous variables, algebraic variables, input variables, and locations are all visualized as rectangles with a thick border. The text in the rectangle is a

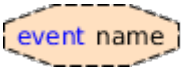
keyword (`event`, `disc`, `cont`, etc), followed by the name of the declaration. For channels (events with a data type), the name is followed by `!?~`.

Event, location, algebraic, and input parameters are visualized as octagons with a thin dashed border. The text in the rectangle is a keyword (`event`, `location`, etc), followed by the name of the parameter. For channels (events with a data type), the name is followed by the event parameter usage restriction flags (e.g. `!` for send, `?` for receive, and/or `~` for synchronization). For channel parameters without flags, `!?~` is used as text after the name of the parameter.

Uses of events and data are visualized as circles. For uses of events by automata (event in the alphabet of the automaton), the circles indicate the type of use: `!` if the automaton sends over the channel, `?` if the automaton receives over the channel, and no text if the automaton synchronizes over the event (the event is in the alphabet of the automaton). For data uses, the circles never contain text.

Different colors are used for declarations, parameters, and uses, as well as for events and data:

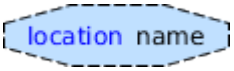
- 

An event declaration
- 

An event parameter
- 

An event usage (automaton sends)
- 

An event usage (automaton receives)
- 
An event usage (automaton synchronizes)
- 

A variable declaration or a location
- 

A location parameter, an algebraic parameter, or an input parameter
- 
A data (variable, location, parameter) usage in a component

Each event usage is coupled to the event declaration or event parameter that it refers to, using a directed or undirected edge. For synchronizing usage an undirected edge is used, for send usage a directed edge from the usage to the event declaration or event parameter is used, and for receive usage a directed edge from the event declaration or event parameter to the usage is used.

Each data usage in a component not related to a declaration (variable, location, or parameter) gets a circular node to indicate usage. It is coupled to the declaration it uses, via a directed edge from the declaration to the use, to indicate the flow of data. Duplicate edges are avoided.

Event parameters of component instantiations are coupled to the actual event (or event parameter) that is provided by the instantiation, using a directed or undirected edge. If the parameter is a channel parameter that explicitly allows sending (has a `!` flag) and doesn't allow receiving (has no `?` flag), a directed edge from the parameter to the actual event is used. If it explicitly allows receiving (has `?` flag) and doesn't allow sending (has no `!` flag), a directed edge from the actual event to the parameter is used. If it explicitly allows both sending and receiving (has both a `!` and a `?` flag), a bidirectional edge is used. In all other cases, an undirected edge is used.

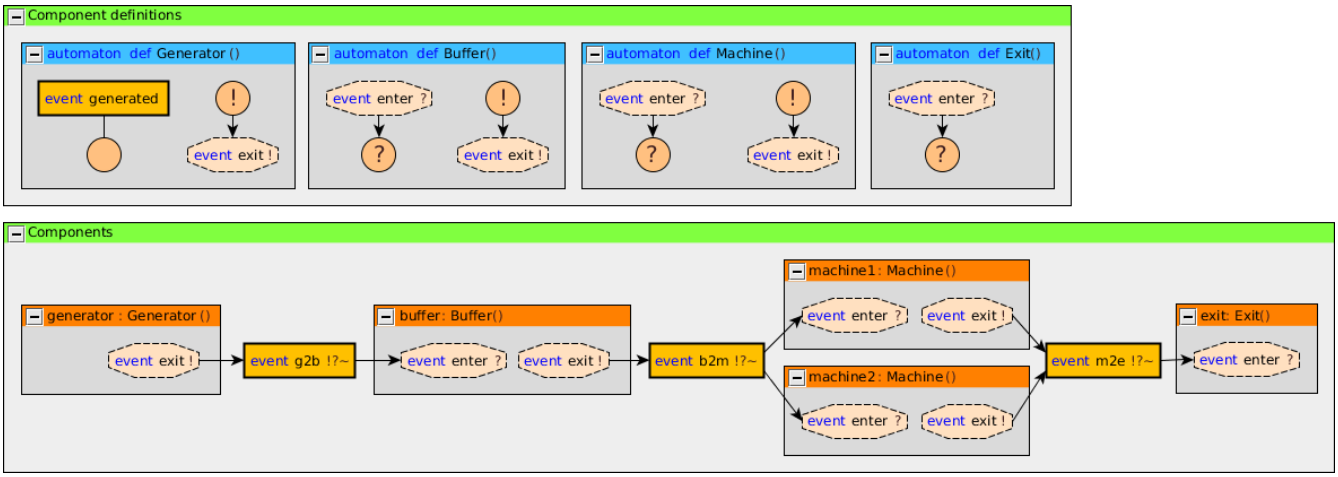
Location parameters of component instantiations are coupled to the actual location (or location

parameter) that is provided by the instantiation, using a directed edge from the actual location to the parameter, to indicate the flow of data. Input parameters are also coupled to the actual input variable (or input parameter) that is provided. Similarly, algebraic parameters are related to the arguments. However, for algebraic parameters, an expression may be provided. For each variable, location, and parameter used in the argument, a directed edge is added. Duplicate edges are avoided.

From declarations (variables, locations, parameters), directed edges are included in the diagram for all relations to other declarations. For instance, if a discrete variable is initialized using the value of an algebraic variable, there will be an edge from the algebraic variable to the discrete variable. Similarly, if in a continuous variable is used in a guard, there will be an edge from the continuous variable to the location that contains the edge. Duplicate edges are avoided.

Relations via annotation argument values are ignored.

As an example consider the following relations diagram of the CIF 'machine/buffer' example, showing only the event relations:



Using the *Relations* option (see the Options section above), certain kinds of relations included or excluded from the diagram. By default all kinds of relations are included. The following kinds of relations are available:

- event relations (`event`)
- data relations (`data`)

In the option dialog, each of the features can be enabled and disabled individually, using a checkbox.

From the command line, using the `-r` or `--relations` option, the names of the different kinds of relations, as indicates between parentheses above, should be used, separated by commas. As an example, consider `-r event` or `--relations=event`. Only the event relations are included, while all the other relations are excluded.

The kinds that are specified using the option replace the default kinds. However, it is also possible to specify additions and removals relative to the default kinds, by prefixing names with a `+` or `-` respectively. For instance, when using `-r -event` or `--relations=-event` the default kinds are used, with the event relations excluded (due to `-event`). Replacements (no prefix) may not be combined

with additions/removals (+ or - prefix).

Specifying a replacement kind type twice leads to a warning being printed to the console. Adding a kind that is already present or removing a kind that is not present, also leads to a warning being printed. Finally, generating a relations diagram without any relations also leads to a warning being printed.

Layout

The generated diagrams can be opened in the yEd Diagram Editor using **File › Open...**. After opening the diagram, all nodes overlap, as the generated diagrams don't contain any positions for them. yEd can automatically layout the diagram. Several layout algorithms are available in the **Layout** menu.

Good results for model diagrams can often be obtained by using the *Orthogonal layout*. Select **Layout › Orthogonal › Classic** to show the *Orthogonal Layout* window. Click the **[Dock]** button to dock the window. Once docked, select *Integrated* for the *Edge Labeling* option (*Labelling* category) and *Free* for the *Edge Label Model* option (*Labelling* category). Also enable the *Consider Node Labels* option (*Labelling* category). Press the *Apply changes* button (the green rectangle) at the top of the docked window to apply the changes to the layout.

You may want to try different values for the *Grid* option (*Layout* category), to get more or less spacing between nodes. Once you change a value, just apply the new layout, as before. The automatic layout algorithms aren't perfect, but they are usually a good starting point.

By default, edge labels have a light gray background. This makes them easier to read when placed on the edges. By changing the *Edge Label Model* option (*Labelling* category) from *Free* to *Side Slider*, labels are not placed on the edges, but besides them. However, this often leads to some overlap, requiring manual layouting. If the *Side Slider* variant is used, you may want to change the background of the edge labels to be transparent. This can be achieved by enabling the *Transparent edge labels* option (see the Options section above).

Good results for relations diagrams can often be obtained using the *One-Click Layout* functionality of yEd. Select **Layout › One-Click Layout** or use the corresponding toolbar button. This option is only available in yEd 3.14 or newer.

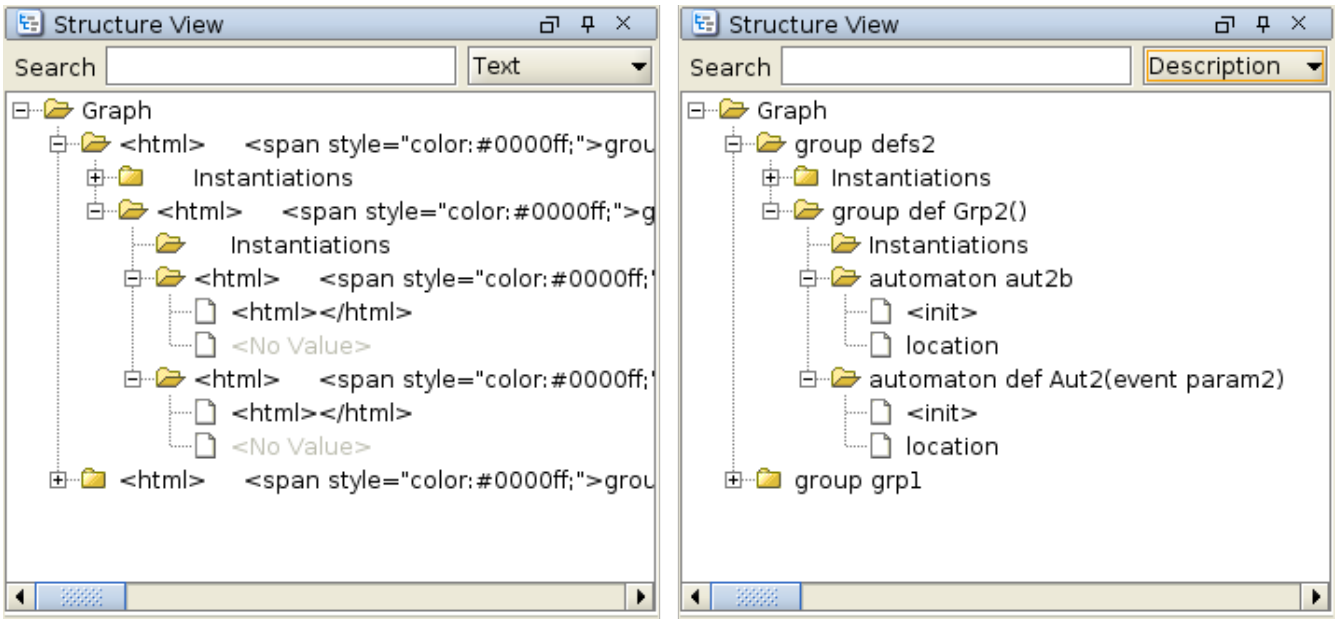
You can also close (collapse) and open (expand) groups and apply the layout again.

yEd has many more options. You can explore them using the included help text, by selecting **Help › Help**.

Structure View

yEd has a *Structure View* window that can be enabled via **Windows › Structure View**. The *Structure View* window can be used to see and navigate the structure of the CIF model. By switching from *Text* to *Description* (see upper right corner of the *Structure View* window), useful descriptions

are shown for all elements. Furthermore, switching also ensures that the HTML code that is used for syntax highlighting is not shown in the *Structure Viewer* (if syntax highlighting is enabled, see the Options section above). The following image shows the difference:



4.7. Scripting

4.7.1. Introduction to scripting

All CIF tools can be used in ToolDef scripts. ToolDef is a cross-platform and machine-independent scripting language that supports command line execution, but is also available as plug-in for Eclipse, providing an integrated development experience. See the ToolDef website for more information on ToolDef.

Scenarios

Using CIF tools in a ToolDef script can be useful for various reasons.

Scripts allow executing multiple tools, one after the other, for instance to perform data-based synthesis on some sub-systems, merge the resulting supervisors, and generate PLC code from it. Once a script is made, these steps can be easily repeated whenever the models for the sub-systems change. This is much easier than starting each of the tools manually, and saves valuable time. Furthermore, it is possible to share your script with others, who can then perform the same steps, and get the same results.

A script can also be used to execute a single tool, specifying the arguments used when executing it. For instance, a script could execute the simulator, specifying not only which model (file) to simulate, but also the simulation options. If various visualizations are enabled, variables are filtered, etc, manually configuring the options for each simulation can become tiresome. By specifying them in a script, simulating again with the same options becomes as simple as executing the script again. By sharing such a script with others, they can simulate the same model with the same options.

Another use of scripts is to repeatedly execute the same tool, but with different arguments. A good

example is repeatedly simulating a model. Instead of manually simulating a model with stochastics hundreds or even thousands of times, which can be very time consuming, a script can be used to automate this task. The script can then also be used to automatically collect the results, and process them. Others can use the same script to repeat the experiments and verify the results, or to perform similar experiments.

Library import

In order to use CIF tools in a ToolDef script, the library with all CIF tools needs to be imported. The import statement to use is:

```
from "lib:cif" import *;
```

This statement imports all tools from the `cif` library, a library registered by the CIF tooling. You'll want to include this import statement in all your own scripts that use CIF tools.

Execution a tool

Here is a simple example of a ToolDef script that simulates a model using the CIF simulator:

```
from "lib:cif" import *;

cifsim("some_model.cif -i auto -t 10");
```

The second statement executes the `cifsim` tool (the CIF simulator). This is just one of the tools that is available, but there are many more.

Command line arguments

Command line arguments can be used to let a tool know on which files it should operate, and what options it should use. The example above uses `"some_model.cif -i auto -t 10"` as arguments, consisting of the model to simulate (the `some_model.cif` file) and some simulation options (automatic simulation due to `-i auto`, and a simulation end time of 10 time units due to `-t 10`).

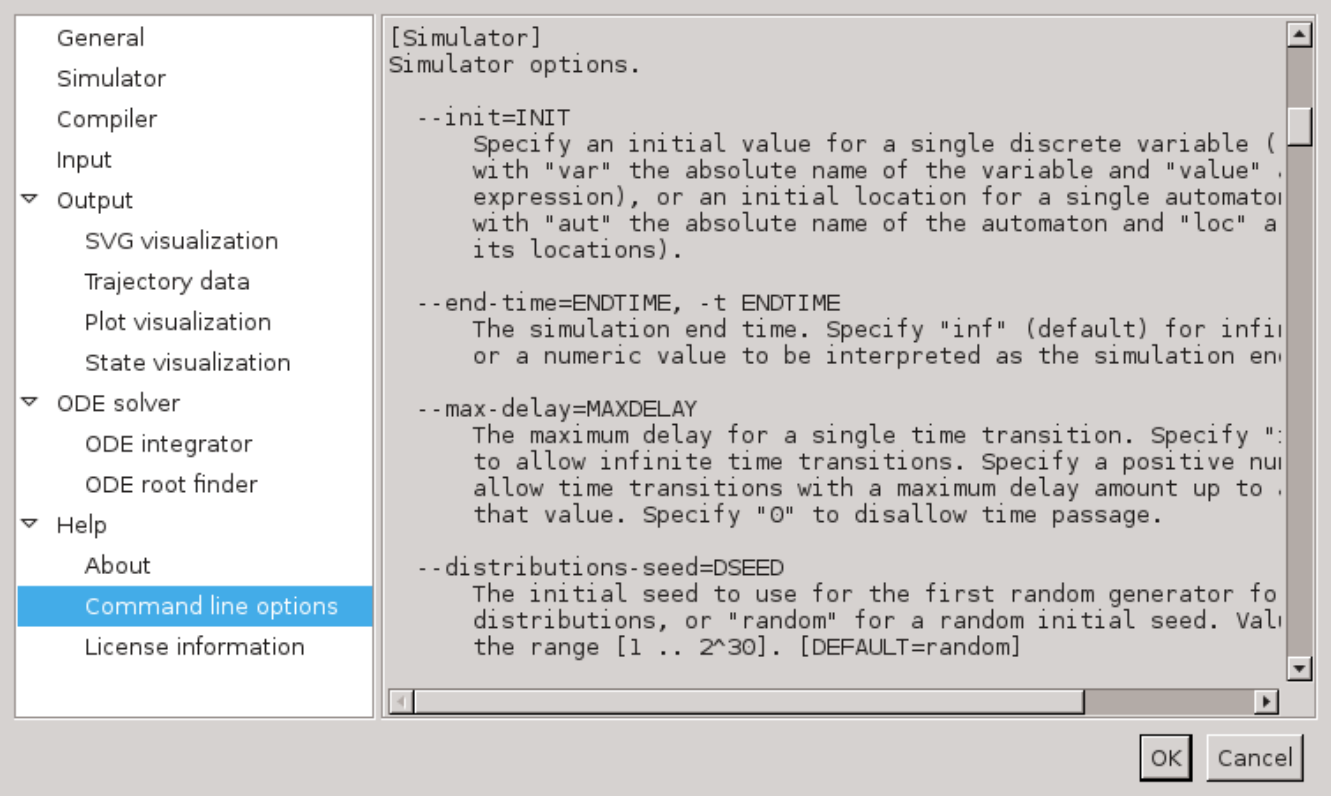
The command line options to use in ToolDef, are identical to the command line options to use for command line scripts, in a shell or command window. The CIF documentation generally refers to command line arguments or command line options, which can thus be used on the actual command line, as well as in ToolDef scripts.

The command line arguments that are available are different for each tool. All CIF tools have `-h` and `--help` command line arguments that can be used to print the application help text to the console. The help text contains detailed information on all the command line arguments supported by the tool. Here is an example of how to show the help text for the simulator:

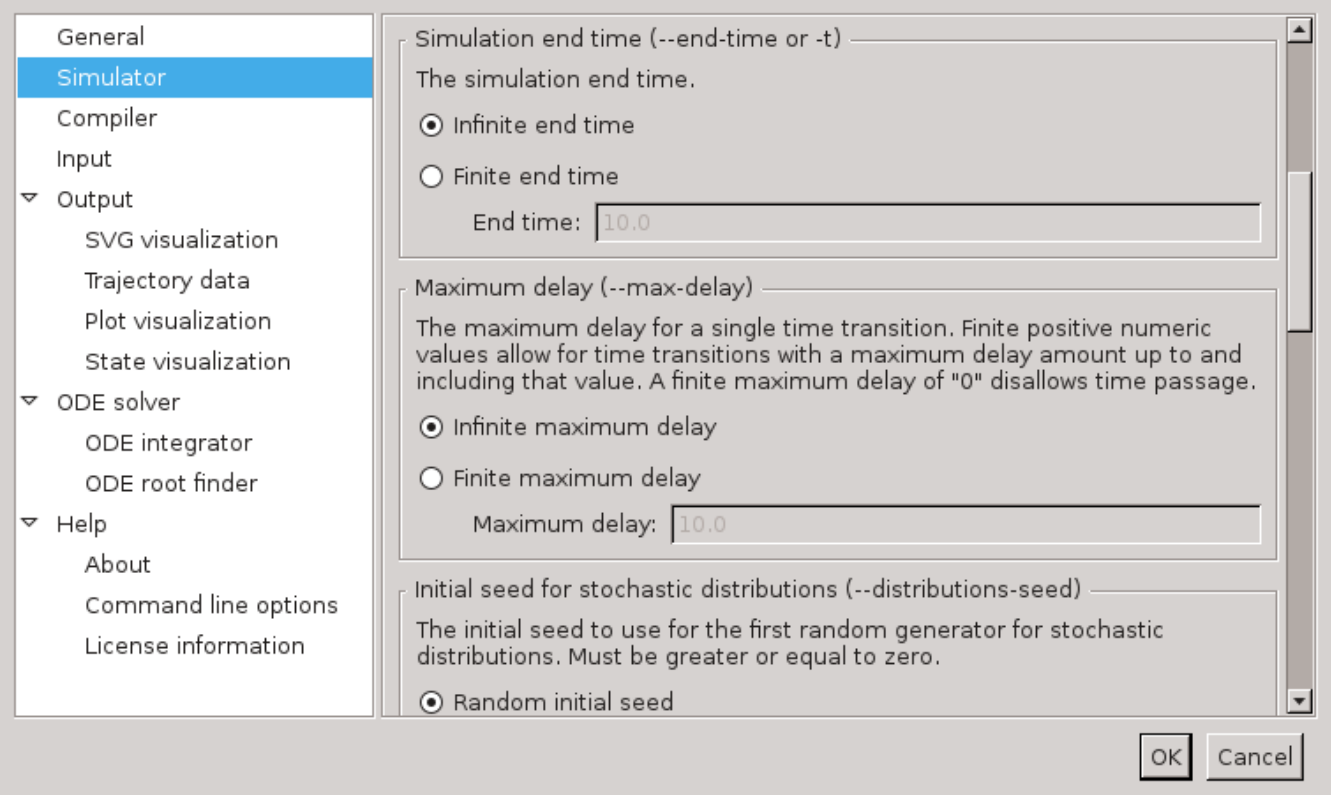
```
from "lib:cif" import *;

cifsim("-h");
```

This information can also be seen in the option dialog:



The option dialog shows the same help text as the command line `--help` and `-h` options. In this screenshot, you can see the help text of the `--end-time` or `-t` option. This particular option can also be configured using the option dialog. It is part of the *Simulator* category:



There, it is listed as *Simulator end time* option. After that name, the command line names of the option (`--end-time` and `-t`) are also listed. This makes it easier to link options in the option dialog with command line options.

In the example ToolDef scripts so far, all command line arguments are provided as a single string of text. However, it is also possible to provide each command line argument as a separate string. In fact, you may provide as many strings as you like, and each string may contain as many command line arguments as you like. For instance, the following statements each execute the simulator, and all have the same command line arguments, but provided in a different way:

```
cifsim("some_model.cif -i auto -t 10");       // Single string, all arguments.
cifsim("some_model.cif", "-i auto", "-t 10"); // Multiple strings, multiple arguments
each.
cifsim("some_model.cif", "-i", "auto", "-t", "10"); // Multiple strings, one argument
each.
```

For simulations, disabling a certain simulation option is a common task. By listing each simulation option on a separate line of the script, disabling or enabling a simulation option becomes as easy as commenting or uncommenting a line of the script. For instance, consider the following script:

```
from "lib:cif" import *;

cifsim(
    "some_model.cif",
    "-i auto",
    "-t 10",
);
```

The model to simulate, the enabling of the automatic input mode, and the simulation end time, are each specified on a separate line. This makes it very easy to disable the simulation end time:

```
from "lib:cif" import *;

cifsim(
    "some_model.cif",
    "-i auto",
    // "-t 10",
);
```

To comment a line or multiple selected lines in a ToolDef text editor, use keyboard shortcut `Ctrl` + `/` or the corresponding toolbar button (). For uncommenting, use keyboard shortcut `Ctrl` + `Shift` + `/` or the corresponding toolbar button ().

Output redirection and other options

Besides the application specific options that can be configured using the command line arguments, all tools such as the `cifsim` tool, have some additional options. This includes the possibility to write the output of the tool to a file instead of the console:

```
from "lib:cif" import *;
```

```
cifsim("some_model.cif -i auto -t 10", stdout="output.txt");
```

The `stdout` parameter of the `cifsim` tool is used to specify the filename of the file to which to write the output. A complete list of the available parameters and their meaning, is available on the Overview of scriptable tools page.

4.7.2. Overview of scriptable tools

This page provides an overview of the CIF tools that are available in the `cif` library that is registered with ToolDef by CIF.

Tools

The following table shows the relation between the names of the ToolDef tools in the library and the tools of the CIF tool set:

ToolDef tool name	CIF tool
`cif2yed`	CIF to yEd transformer
`cifinfogen`	CIF information generator
`cifdatasynth`	Data-based supervisory controller synthesis
`cifmultilevel`	CIF multi-level splitter
`cifcontrollercheck`	Controller properties checker
`cif2supremica`	CIF to Supremica transformer
`cifsim`	CIF simulator
`cif2mcrl2`	CIF to mCRL2 transformer
`cif2uppaal`	CIF to UPPAAL transformer
`cifcodegen`	CIF code generator
`cif2plc`	CIF PLC code generator (deprecated)
`cifplcgen`	CIF PLC code generator (stable)
`cifrelations`	CIF relations collector
`cif2cif`	CIF to CIF transformer
`cifmerge`	CIF merger
`cifevtdis`	CIF event disabler
`cifexplorer`	CIF explorer
`cifprod`	Event-based synchronous product
`cifsupsynth`	Event-based supervisor synthesis
`cifsynthanalys`	Event-based synthesis analysis
`cifncchk`	Event-based nonconflicting check

ToolDef tool name	CIF tool
`cifctrlchk`	Event-based controllability check
`ciflngeqv`	Event-based language equivalence check
`cifnfadfa`	Event-based NFA to DFA automaton conversion
`cifdfamin`	Event-based DFA minimization
`cifproj`	Event-based automaton projection
`cifobschk`	Event-based observer check
`cifabstr`	Event-based automaton abstraction
`ciftrimchk`	Event-based trim check
`ciftrim`	Event-based trim

Parameters

For every CIF tool, two ToolDef tool variants are available in the `cif` ToolDef library. All the CIF tools in the `cif` ToolDef library have the same tool signatures, including the same parameters. Only their names differ. Below the tool signatures for the two variants are listed, with `toolname` instead of the actual tool name (e.g. `cifsim`):

```
tool int toolname(
            string... args,
            string stdin = "-",
            string stdout = "-",
            string stderr = "-",
            bool appendOut = false,
            bool appendErr = false,
            bool errToOut = false,
            bool ignoreNonZeroExitCode = false
)

tool int toolname(
            list string args = [],
            string stdin = "-",
            string stdout = "-",
            string stderr = "-",
            bool appendOut = false,
            bool appendErr = false,
            bool errToOut = false,
            bool ignoreNonZeroExitCode = false
)
```

The meaning of each of the parameters is as follows:

args

The command line arguments of the application. Each argument string is parsed to zero or more actual arguments. It is possible to use a single string with all arguments (where the arguments

themselves are separated by spaces), comma separated strings for each of the arguments (each string has one argument), or a mix of those.

In argument strings, individual arguments are separated by whitespace (spaces, tabs, new lines, etc). The whitespace itself is ignored, and only serves as separation. To include whitespace in an argument, the argument (or a part of it), may be quoted, by putting it between double quotes (`"`) or single quotes (`'`). Characters may be escaped by prefixing them with a backslash (`\`). This is particularly useful for single/double quotes, and escape characters, and can also be used to escape spaces. Escapes work the same inside of quoted parts as they do outside of quoted parts.

stdin

Specify whether to have a standard input (stdin) stream and where the input comes from. Use `""` to not have a stdin stream, "-" to use the stdin stream of the ToolDef interpreter, or otherwise an absolute or relative local file system path of the file from which to read the standard input. May contain both `\` and `/` as file separators.

stdout

Specify whether to have a standard output (stdout) stream and where to write the standard output. Use `""` to not have a stdout stream, `"-"` to use the stdout stream of the ToolDef interpreter, or otherwise an absolute or relative local file system path of the file to which to write the standard output. May contain both `\` and `/` as file separators.

stderr

Specify whether to have a standard error (stderr) stream and where to write the standard error output. Use `""` to not have a stderr stream, `"-"` to use the stderr stream of the ToolDef interpreter, or otherwise an absolute or relative local file system path of the file to which to write the standard error output. May contain both `\` and `/` as file separators. Is ignored when the standard error stream is redirected to the standard output stream.

appendOut

Whether to append to the stdout file (`true`) or overwrite it (`false`). Is ignored when standard output is not written to a file.

appendErr

Whether to append to the stderr file (`true`) or overwrite it (`false`). Is ignored if standard error output is not written to a file. Is also ignored when standard error stream is redirected to the standard output stream.

errToOut

Whether to redirect the standard error stream to the standard output stream (`true`) or use separate streams (`false`).

ignoreNonZeroExitCode

Whether to ignore non-zero exit codes (`true`) or consider them as errors (`false`).

Return value

All the tools return an integer value. This value is the exit code of the application. A zero (`0`) exit

code means the application terminated without any errors. A non-zero exit code means the application terminated with an error.

Implementation

All the tools are internally implemented using the ToolDef `app` tool.

Chapter 5. CIF examples

CIF ships with several examples and benchmark models. To obtain these examples, follow these steps:

- Start the Eclipse ESCET IDE.
- Click **File › New › Example...**.
- Select *CIF Examples* or *CIF Benchmarks* and click **[Next]**.
- Choose the name of the new project that will be created.
- Choose a location for the new project, on your hard disk. If you leave the *Use default location* option enabled, the new project will be created in workspace directory. See the Eclipse ESCET documentation for more information on workspaces.
- Click **[Finish]** to create the new project, and fill it with the examples.

For new releases of CIF, the examples may change. The examples project that you previously created won't be automatically updated. You can however go through the steps again, and get a second project, with the new examples. The version of the release is included in the suggested project name, allowing multiple example projects for different releases.

Most examples also include a ToolDef script (`.tooldef` file) that can be used to perform various tasks, such as simulation. To execute such a script, right click a `.tooldef` file an choose **Execute ToolDef**. Alternatively, select the file and press `F10`.

For the following examples, additional information is available:

- Supervisory controller synthesis FIFO example

Chapter 6. CIF release notes

The release notes for the versions of CIF and the associated tools, as part of the Eclipse ESCET project, are listed below in reverse chronological order.

See also the Eclipse ESCET toolkit release notes covering those aspects that are common to the various Eclipse ESCET tools.

6.1. Version 13.0

TBD

CIF language

- CIF now features list comprehension expressions. See the CIF language tutorial lesson about lists for more information. This new feature is supported by various CIF tools (issues #1288 and #1448).
- Reachability requirement annotations now support optional `from`, `avoid`, `stay` and `bound` arguments for conditional, restricted and bounded reachability requirements (issue #1434).

CIF type checker

- The type checker now warns when a channel has no sender or receiver, unless the specification is marked with a partial specification annotation (issue #1460).
- The type checker has an improved error message for arguments of an `@doc` annotation that can't be evaluated (issue #1288).

CIF simulator

- The simulator no longer crashes when hovering interactive SVG elements that have no outline (shape), such as an empty text box (issue #1450).

CIF data-based synthesis

- The tool now supports conditional, restricted and bounded reachability requirements (issue #1434).
- The tool has a new 'BDD hyper-edge graph creation algorithm' option to specify which algorithm to use for creating the graphs that represent relations between BDD variables in the CIF specification. The graph creation algorithm primarily affects the time it takes to create the graph itself, which can greatly improve performance for larger models, especially models with denser variable relations. However, it can also affect the variable order, as it influences the order in which variable ordering algorithms consider the graph edges (issue #1464).
- The documentation page about fixed-point computations order has some small fixes (issue #1434).

CIF code generator

- Code can now be generated for CIF models that use the `+` binary operator to concatenate arrays (issue #1459).
- Proper code is now generated for the `sign` standard library function on real numbers, when generating C89, C99, JavaScript, HTML or Simulink code (issue #1432).

- The generated C89, C99 and Simulink code has some typo fixes in the function documentation of generated array/tuple type printing code (issue #1459).
- For all target languages, the precondition violation messages now include the specific target language name, making it clearer that these are the violations for this target and that it may differ for other targets (issue #1448).
- An improved error message is shown when arguments of an `@doc` annotation can't be evaluated (issue #1288).

CIF PLC code generator (stable)

- An improved error message is shown when arguments of an `@doc` annotation can't be evaluated (issue #1288).

CIF language tutorial

- The lesson about lists now has additional section headings (issue #1288).
- The lesson about reachability requirements has some small fixes (issue #1434).

CIF language reference manual

- The page about reachability requirements has some small fixes (issue #1434).

6.2. Version 12.0 (2026-06-30)

CIF simulator

- The simulator no longer crashes for models with many thousands of algebraic variables (issue #1420).
- The 'SVG input outline filters' option now works correctly for filters with a `.` in them (issue #715).
- Improved the description of the 'SVG input outline filters' option (issue #715).

CIF to CIF transformer

- The CIF transformer has a new 'Eliminate channels' CIF to CIF transformation (issue #1393).

CIF data-based synthesis

- The tool now supports CIF models with multi-assignments and partial variable assignments in internal user-defined functions (issue #1218).
- The tool now checks up front, before the actual synthesis computation, whether the CIF specification has any reachability requirements that, by themselves or in combination with state invariants, lead to an empty supervisor (issue #1296).
- The tool's up front checks for an empty supervisor, before the actual synthesis computation, now produce more consistent warning messages (issue #1296).
- Various improvements to the documentation, mostly related to options and edge ordering (issues #1408 and #1409).

CIF controller properties checker

- For the bounded response, confluence and non-blocking under control checks, the tool now supports CIF models with multi-assignments and partial variable assignments in internal user-defined functions (issue #1218).

CIF code generator

- The HTML code generator now has an 'SVG input outline style' option, to specify the outline style for interactive SVG elements in generated HTML pages. By default, a red hover and green click color are used, with an outline width of 1.5 pixels (issue #715).
- The HTML code generator now has an 'SVG input outline filters' option, to selectively disable hover/click outlines for interactive SVG elements (issue #715).
- Generated HTML pages now include the SVG images in sorted order, based on their filename (or absolute path in case of identical filenames), to ensure their order can be easily understood and influenced by users (issue #1398).
- Generated HTML pages now specify hover stroke widths using an explicit pixel unit and this width now applies correctly, regardless of any self/ancestor transforms (issue #715).
- Generated HTML pages now have proper hover outlines for interactive SVG elements, where more deeply nested parts of the element could previously lack an outline (issue #715).
- Generated HTML pages have been fixed to ensure that rapid/double clicking interactive elements no longer selects text (issue #715).
- Generated HTML pages now have reduced CSS code duplication through class tagging (issue #715).
- Improved HTML code generation options documentation (issue #715).
- Generated C and Simulink code now consistently uses `FALSE` instead of `false`, preventing compile errors related to `stdbool.h` on some systems (issue #1425).

CIF PLC code generator (stable)

- The tool now has an improved error message for I/O table entries where (part of the) absolute name of the corresponding CIF object is not found in the root of the CIF specification (issue #1402).

6.3. Version 11.0 (2026-03-31)

CIF type checker

- The CIF type checker now warns when a CIF model refers to an SVG image that has a text element with a `shape-inside` style property, an unsupported SVG version 2 feature (issue #1293).

CIF information generator

- The CIF toolset now includes a new tool, the CIF information generator, which can be used to generate information about a CIF specification. The first version generates event participants information and event-enablement conditions. See the tool's documentation for more information (issue #1378).

CIF simulator

- The SVG visualizer and plot visualizer now no longer scale on hi-DPI displays on Windows. This prevents severe performance regressions observed in ESCET v10.0. ESCET v10.0 adopted the Eclipse IDE 2025-09, which comes with improved hi-DPI support on Windows, which caused the regressions (issue #1334).
- The SVG input mode now supports resetting the simulation, by right clicking the SVG

visualizer and choosing the relevant menu item (issue #1338).

- The simulator has a new 'SVG input outline style' option to configure the style to use for the outline when hovering or clicking an interactive SVG element. It can be used to configure the hover and click colors as well as the stroke width. It can also be used to disable such outlines (issue #1352).
- The simulator now uses a stroke width of 1.5 pixels for the outline when hovering or clicking an interactive SVG element, and properly compensates for scale factors when displaying such an outline (issue #1352).
- The simulator now has an 'SVG input outline filters' option to selectively disable hover/click outlines for interactive SVG elements (issue #1352).
- The SVG visualizer now uses the SVG filename, without path and file extension, as its tab name. This allows users to fully customize the tab names and more easily distinguish multiple SVG visualizer tabs (issue #1351).
- The SVG visualizer no longer hangs when showing SVG files with inline images encoded using base64 that contain spaces (issue #1324).
- The 'Debug output' option now additionally allows for CIF/SVG real-time timing debug output to be printed to the console (issue #1355).
- The simulator has greatly improved performance in generating code and compiling it, leading to faster start times of the simulator, for CIF models with many events and automata, where most automata don't have any state/event exclusion invariants in their locations (issue #1396).
- The simulator has improved performance and robustness for writing generated code files, when the 'Debug output' option is configured to write the code generated for the CIF model to disk (issue #1394).
- The simulator documentation has multiple small improvements and extensions (issues #1293, #1346 and #1352).

CIF explorer

- The CIF explorer no longer crashes when exploring CIF models with assignments to tuples where the projection index is a field reference (issue #1379).

CIF data-based synthesis

- The tool now supports CIF models with tuple-typed variables, tuple-typed expressions, multi-assignments on edges, partial variable assignments on edges, and internal user-defined functions that have multiple return types (issue #1218).
- Slightly improved performance of the conversion of the CIF specification to the internal BDD representation, preventing a linear search for when synthesis variables need to be found for CIF objects (issue #1218).
- Slightly improved performance of the conversion of the CIF specification to the internal BDD representation, on average for the CIF benchmarks, by using the reverse variable order for variables assigned on edges (issue #1218).
- The tool now uses smaller BDD variable support sets for transition relations. This may affect performance, although the effect differs per model and the difference is generally minor (issue #1365).

- The tool now converts updates on edges in a different way, which may impact the performance of the conversion of CIF models to the internal BDD representation. The effect differs per model, but the difference is generally minor (issue #1218).
- The debug output for variable ordering has been fixed, for BDD variable indices of interleaved synthesis variables (issue #1360).
- Improved the performance for printing assignments of synthesis edges in debug output (issue #1218).
- The documentation about variable ordering has several small improvements (issue #1367).
- The documentation about supported specifications now lists receiving values of supported types as supported. This was already supported, but is now properly documented (issue #1367).

CIF multi-level splitter

- Fixed missing filenames of partial specifications in generated ToolDef scripts, for partial specifications that have only partially been checked for being supported by data-based synthesis (issue #1337).
- The tool has improved performance and robustness for writing generated ToolDef scripts to disk (issue #1394).
- The tool's documentation has some small improvements and additions (issue #1310).

CIF relations collector

- The CIF toolset now includes a new tool, the CIF relations collector, which analyzes a CIF specification for relations between its various elements and produces three Domain Mapping Matrices (DMMs) with the results. See the tool's documentation for more information (issue #1310).

CIF to UPPAAL transformer

- The transformation now generates a proper `false` guard for non-channel events that are not in the alphabet of the specification (issue #1354).

CIF to mCRL2 transformer

- The transformer has improved performance and robustness for writing generated mCRL2 files to disk (issue #1394).

CIF controller properties checker

- For the bounded response, confluence and non-blocking under control checks, the tool now supports CIF models with tuple-typed variables, tuple-typed expressions, multi-assignments on edges, partial variable assignments on edges, and internal user-defined functions that have multiple return types (issue #1218).
- For the bounded response, confluence and non-blocking under control checks, the tool now has slightly improved performance of the conversion of the CIF specification to the internal BDD representation, preventing a linear search for when synthesis variables need to be found for CIF objects (issue #1218).
- For the bounded response, confluence and non-blocking under control checks, the tool now has slightly improved performance of the conversion of the CIF specification to the internal BDD representation, by using the reverse variable order for variables assigned on edges (issue #1218).

- For the bounded response, confluence and non-blocking under control checks, the tool now uses smaller BDD variable support sets for transition relations. This may affect performance, although the effect differs per model and the difference is generally minor (issue #1365).
- For the bounded response, confluence and non-blocking under control checks, the tool now converts updates on edges in a different way, which may impact the performance of the conversion of CIF models to the internal BDD representation. The effect differs per model, but the difference is generally minor (issue #1218).
- For the bounded response, confluence and non-blocking under control checks, the tool now has improved performance for printing assignments of edges in debug output (issue #1218).
- The documentation about supported specifications had layout issues that have been fixed (issue #1367).
- The documentation about supported specifications now lists receiving values of supported types as supported, for the bounded response, confluence and non-blocking under control checks. This was already supported, but is now properly documented (issue #1367).

CIF code generator

- Generated JavaScript code and HTML pages now allow logging algebraic variables as part of logging states. It is disabled by default (issue #1381).
- The HTML code generator now has a new 'HTML speeds' option, to configure the minimum, default and maximum speed for the execution of the generated HTML page. Generated HTML pages now have a slider to configure the speed (within the configured range), in their 'settings' menu (issue #715).
- Some error messages of the 'HTML frequencies' option have textual fixes (issue #715).
- Generated HTML pages have 'settings' menus with improved positioning and layout (issue #715).
- Generated HTML pages now have their frequency and 'show SVG image' configuration inside the 'settings' menu (issue #715).
- Generated Simulink code has been fixed, for time-dependent guard evaluation functions, which could have duplicate names, leading to compile errors (issue #1384).
- Small improvements and a textual fix in the code generator documentation (issue #715).

CIF PLC code generator (stable)

- Fixed a bug relating to generating code for tuple projections that use field names, which could occur under certain circumstances (issue #1366).
- The generator has improved performance and robustness for writing generated PLC code to disk (issue #1394).

CIF PLC code generator (deprecated)

- The generator has improved performance and robustness for writing generated PLC code to disk (issue #1394).

CIF examples

- CIF FIFO example: The ToolDef scripts to check that the manually-written and synthesized supervisors are language equivalent, have several small improvements. Additionally, the script that uses mCRL2 for the check has been fixed (issue #769).

CIF website

- The CIF history page had some broken links that have now been fixed (issue #1386).

CIF documentation

- Small improvements and a textual fix for the FIFO example in the SBE in practice documentation (issue #769).
- The CIF tutorial lesson on 'Inkscape' now documents that text areas are not supported, and the lesson also has some slight textual improvements (issue #1293).
- The CIF reference manual now documents that CIF supports only SVG 1.1, not SVG 2, in the documentation about SVG file declarations (issue #1293).

6.4. Version 10.0 (2025-12-19)

CIF language

- CIF now features a `requirement:reachable` annotation, to specify reachability requirements, which can be used to specify stronger liveness requirements in combination with marking. See the documentation for more information (issue #1296).

CIF type checker

- The type checker no longer crashes on checking for effectively-constant variables in automata with at least one discrete variable and at least one invalid addressable in an assignment of an edge of the automaton (issue #1295).

CIF simulator

- The CIF simulator SVG visualizer has been fixed for handling of vendor-specific CSS properties starting with a `-`, such as Inkscape CSS properties, in `style` attributes of SVG elements. Vendor-specific properties are now removed to ensure that the Batik SVG library, used to render SVG images, properly processes the properties that follow the vendor-specific properties (issue #886).
- The CIF simulator SVG visualizer now properly handles and reports CSS rendering errors, similar to SVG XML/DOM errors (issue #888).
- The trace input mode documentation has been slightly improved (issue #1285).

CIF data-based synthesis

- The tool now supports reachability requirements, ensuring a correct-by-construction controlled system where every reachable state adheres to all these requirements (issue #1296).
- The tool now uses the 'Linearize (product with merge)' CIF to CIF transformation instead of the 'Linearize (product)' CIF to CIF transformation, for the creation of linearized hyper edge relations for variable ordering, and when converting edges to BDD transition relations. For some models, this can prevent exponential blow-ups in the number of linearized edges. On average, for the CIF benchmarks, there is a 12% reduction in the maximum number of nodes and a 45% reduction in the number of operations. However, the results differ per model, as is often the case with changes that affect variable ordering heuristics (issue #1203).
- The tool has reduced memory usage for conversion of linearized edges to BDDs, by lazily creating and converting them one by one, rather than first creating all of them before they

are converted to BDDs (issue #1203).

- The tool now prevents duplicate work and, in certain situations, even prevents exponential blow-ups during the computation of variable relations by hyper-edge creators when collecting variables and/or comparison binary expressions (issue #1303).
- The tool now no longer collects duplicate comparison binary expression instances in the legacy hyper-edge creator. This may affect variable relations and therefore variable ordering, although the CIF benchmarks are not affected (issue #1303).
- The tool's 'Fixed-point computations order' option now requires that the names of the fixed-point computations are separated by commas, rather than by dashes. This is a backwards incompatible change. If the old option syntax is used, a warning is printed (issue #1296).
- The tool's 'Fixed-point computations order' option now requires specifying four instead of three computations, to include the new fixed-point computations for the reachability requirements. For the old option syntax, the new fourth computation cannot be specified, and is always performed after the non-blockingness computation. For more information, see the tool documentation (issue #1296).
- The tool now prints the configured statistics even if synthesis results in an empty supervisor (issue #1284).
- The tool now prints the fixed-point computations timings in the order that the fixed-point computations are configured and performed, rather than always printing them in the default order (issue #1289).
- The tool's debug output for fixed-point computations has improved terminology (issue #1296).
- The tool's timing statistics now use better terminology for the fixed-point computation timings (issue #1296).
- The tool's 'BDD cache' statistic has been renamed to 'BDD node and operation' statistic. The tool's `bdd-perf-cache` statistic name has similarly been renamed to `bdd-perf-node-op`. For backward compatibility, the old statistic name can still be used for the option, but will lead to a warning. Some of the node creation counts that are printed for this statistic have been renamed as well. Furthermore, the order of the node creation counts has changed in the printed output (issue #1304).
- The tool's performance documentation page has been extended with some easy-to-use high-impact recommendations to improve synthesis time (issue #1290).
- The tool's documentation page about statistics has various improvements and fixes related to the BDD node and cache statistic, has improved terminology use (BDDs are graphs, not trees), and is clearer about the 'Maximum used BDD nodes' statistic being very costly (issue #1304).
- The tool's documentation page about operation caches has multiple small improvements (issue #1304).
- The tool's documentation now links from the forward reachability page to the fixed-point computations order page, pointing out that it is by default performed last (issue #1290).
- The tool's option documentation has been slightly improved (issue #1290).
- The tool's documentation now has an extra paper reference that provides further details on the synthesis algorithm and how it handles the various CIF language constructs (issue #1296).

CIF to CIF transformer

- A new CIF to CIF 'Linearize (product with merge)' transformation has been added. It preserves all behavior like 'Linearize (product)', but prevents blow-ups by merging linearized edges where possible (so, if it can statically detect that there is no non-determinism) like 'Linearize (merge)'. Compared to 'Linearize (product)' this can in certain cases prevent exponential blow-ups in the number of linearized edges (issue #1203).
- The CIF to CIF 'Convert to interface' transformation now generates interface specifications with an `@@partial` annotation (issue #1280).
- The CIF to CIF 'Eliminate state invariant' transformation no longer eliminates requirement state invariants (issue #1188).
- The CIF to CIF 'Remove requirements' transformation now removes all reachability requirement annotations from the specification (issue #1296).
- Small improvements/fixes for the documentation of the CIF to CIF 'Linearize' transformations (issue #1203).

CIF to mCRL2 transformer

- The transformer no longer supports requirement state invariants (issue #1188).
- The transformer now uses the 'Linearize (product with merge)' CIF to CIF transformation instead of the 'Linearize (product)' CIF to CIF transformation, in certain cases preventing exponential blow-ups in the number of linearized edges (issue #1305).

CIF controller properties checker

- For the bounded response, confluence and non-blocking under control checks, the tool now uses the 'Linearize (product with merge)' CIF to CIF transformation instead of the 'Linearize (product)' CIF to CIF transformation, for the creation of linearized hyper edge relations for variable ordering, and when converting edges to BDD transition relations. For some models, this can prevent exponential blow-ups in the number of linearized edges. The results differ per model, as is often the case with changes that affect variable ordering heuristics (issue #1203).
- For the bounded response, confluence and non-blocking under control checks, the tool has reduced memory usage for conversion of linearized edges to BDDs, by lazily creating and converting them one by one, rather than first computing all of them before they are converted to BDDs (issue #1203).
- For the bounded response, confluence and non-blocking under control checks, the tool now prevents duplicate work and, in certain situations, even prevents exponential blow-ups during the computation of variable relations by hyper-edge creators when collecting variables and/or comparison binary expressions (issue #1303).
- For the bounded response, confluence and non-blocking under control checks, the tool now no longer collects duplicate comparison binary expression instances in the legacy hyper-edge creator. This may affect variable relations and therefore variable ordering (issue #1303).

CIF code generator

- The Simulink code generation documentation has been slightly improved (issue #1285).

CIF merger

- The CIF merger now has an option to disable the removal of controller properties annotations of the input specifications, allowing these annotations to be included in the merged specifications. This option must be used with care, to ensure the annotations remain valid in the merged specifications. Consult the tool documentation for more information before using this new option (issue #1316).

CIF examples

- Improved the CIF `button_lamp_timer` example to use decreasing timers instead of increasing ones, generate code using the new PLC code generator, perform controller properties checks before generating PLC code, and have improved console output from the PLC code generation script (issue #1316).

CIF benchmarks

- The benchmarks overview generation script now has improved handling and reporting of missing values (issue #1284).
- The 'bridge' benchmark model has some whitespace improvements (issue #1285).

CIF language tutorial

- The annotations lesson has been split into multiple lessons (issue #1297).
- The section about language extensions has been renamed to a section about synthesis, and the synthesis lesson has been split into multiple lessons (issue #1302).

CIF documentation

- The CIF language tutorial lesson on synthesis has some fixes, textual improvements and small extensions (issue #1291).
- The CIF documentation no longer contains zero width space characters generated by AsciiDoc, ensuring that there are no undesired line breaks caused by them (issue #1279).
- The CIF documentation no longer links to `index.html` in link URLs, but instead now links to the parent that contains the `index.html`, resulting in nicer URLs (issue #1317).
- The CIF history page has an updated paper reference (issue #1319).
- Where relevant, the documentation now refers to the recommended PLC code generator rather than to the deprecated one (issue #1315).

6.5. Version 9.0 (2025-09-30)

CIF language

- CIF now features the `@@partial` built-in annotation. It can be added to partial specifications to suppress certain warnings that are often false positives for partial specifications. See the language tutorial and language reference manual for further information (issue #995).

CIF simulator

- The trace input mode now supports a new type of command in trace files: input commands. An input command allows assigning a new value to an input variable during simulation (issue #710).
- The plot visualizer no longer crashes if the specification has no initial state (issue #1250).

- The plot visualizer is no longer shown if the specification has no initial state (issue #1250).

CIF to CIF transformer

- The *Eliminate state invariants* CIF to CIF transformation no longer supports synchronizing events, events that are in the alphabet of multiple automata. You can eliminate synchronization by applying the *Linearize (product)* CIF to CIF transformation, before applying the *Eliminate state invariants* transformation (issue #1247).

CIF merger

- The tool now has an option to control whether the resulting merged specification is a partial specification. By default, a merged specification is non-partial, but this can be changed by enabling the option (issue #995).

CIF data-based synthesis

- The tool now supports CIF models with multiplication (binary `*` operator) and `if` updates (issue #1216).
- The tool now supports internal user-defined functions and calls to such functions (with supported arguments), provided that the functions satisfy some conditions: they have exactly one return type; they don't use recursive calls; they don't contain `while` statements; they don't perform assignments that may result in runtime out-of-range errors; they are only used in function calls (so not used as data); and only use boolean, ranged integer and enumeration types as parameter and return types, and as types of their local variables (issue #1237).
- The tool now preserves valid cache entries when performing BDD garbage collection, improving synthesis performance (issue #1241).
- The tool now has additional caching, to prevent duplicate work, and in certain situations even prevent exponential blow-ups. This concerns: the result of the pre-condition check for supported expression forms. It was already cached for references to algebraic variables being checked in predicates, but not yet in other expressions (issue #1233).
- The tool now has improved performance for BDD variable ordering of models with many linearized edges and variable relations (issue #1258).
- For the `nodes` BDD output mode, now only ranged integer types are used to represent BDD variable and node indices (issue #1238).
- The tool no longer crashes on using the `sign` function on a `real`-typed value. It is still unsupported (except if it can be statically evaluated), but it now properly reports the problem rather than crashing (issue #1216).
- The *BDD library initial node table size* option now requires a size that is at least 2, preventing crashes (issue #1246).
- Improved the tool documentation for multi-assignments and partial-variable assignment constraints, using similar texts as in the CIF controller properties checker documentation (issue #1216).
- Removed unnecessary precondition constraints that are implied by other constraints, from the tool documentation (issue #1216).

CIF event-based tools

- The event-based language equivalence check has been fixed for the only difference being in the marking of initial states, or the language from initial states. Previously, this was not

detected, leading the tool to wrongly conclude that the specifications were language equivalent (issue #1264).

- The event-based language equivalence check has improved output in case the counter example concerns a pair of initial states (issue #1264).
- The event-based language equivalence check and DFA minimization have improved readability of partitioning blocks printed as part of debug output (issue #1264).

CIF controller properties checker

- The tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with multiplication (binary `*` operator) and `if` updates (issue #1216).
- The tool now supports, for the bounded response, confluence and non-blocking under control checks, internal user-defined functions and calls to such functions (with supported arguments), provided that the functions satisfy some conditions: they have exactly one return type; they don't use recursive calls; they don't contain `while` statements; they don't perform assignments that may result in runtime out-of-range errors; they are only used in function calls (so not used as data); and only use boolean, ranged integer and enumeration types as parameter and return types, and as types of their local variables (issue #1237).
- The tool now preserves, for the bounded response, confluence and non-blocking under control checks, valid cache entries when performing BDD garbage collection, improving synthesis performance (issue #1241).
- The tool now has additional caching, for the bounded response, confluence and non-blocking under control checks, to prevent duplicate work, and in certain situations even prevent exponential blow-ups. This concerns: the result of the pre-condition check for supported expression forms. It was already cached for references to algebraic variables being checked in predicates, but not yet in other expressions (issue #1233).
- The tool now has, for the bounded response, confluence and non-blocking under control checks, improved performance for BDD variable ordering of models with many linearized edges and variable relations (issue #1258).
- The tool, for the bounded response, confluence and non-blocking under control checks, no longer crashes on using the `sign` function on a `real`-typed value. It is still unsupported (except if it can be statically evaluated), but it now properly reports the problem rather than crashing (issue #1216).
- Removed unnecessary constraints that are implied by other constraints, from the tool documentation, for the bounded response, confluence and non-blocking under control checks (issue #1216).

CIF to mCRL2 transformer

- The transformer now applies the *Eliminate state invariants* CIF to CIF transformation as preprocessing after linearization, rather than before linearization (issue #1247).

CIF code generator

- Fixed JS/HTML code generated for real division (`/`) on integer arguments where the divisor is never zero (issue #1242).
- Fixed real overflow error message of the `divide` utility function for generated JS/HTML code

(issue #1242).

- Generated JS/HTML code for the `divide` utility function has JSDoc improvements (issue #1242).
- Small improvement for comments in generated JS/HTML code (issue #1262).

CIF PLC code generator (stable)

- The tool now has improved precondition checking, skipping to report certain precondition violations if they are already covered by other violations (issue #1255).

CIF benchmarks

- The `wafer_scanner` benchmark has improved tuned variable order settings (issue #1234).

CIF website

- The CIF website home page now shows some news teasers and links to the new 'News' page with all the news (issue #982).
- The CIF website home page now has a more-prominent SBE course link (issue #1240).
- The CIF website home page now links to the 'Community' page (issue #982).
- The CIF history page has been slightly improved (issue #982).

CIF documentation

- The documentation about built-in annotations, in the CIF tutorial and CIF language reference manual, no longer categorizes the different annotations (issue #995).
- The documentation pages for the different built-in annotations in the CIF language reference manual now have additional section headings (issue #995).
- The 'Annotation' section of the *Merge enumerations* CIF to CIF documentation has been fixed (issue #1253).
- The documentation has various small textual improvements (issue #1262).

6.6. Version 8.0 (2025-06-30)

CIF general

- CIF has a new logo and icon. The logo and icon have been updated on the CIF website, in the CIF documentation, and in the UI of the ESCET IDE (issue #1143).
- All CIF tools can now be started in the ESCET IDE from right click menus of `.cifx` files and editors (issue #1229).
- The documentation of all CIF tools now indicate that the tools can be started from `.cifx` files (issue #1229).
- Various screenshots of CIF in the ESCET IDE, as part of the CIF documentation, CIF website and CIF examples, have been updated to include the ESCET icon introduced in release v3.0 (issue #1197).

CIF language

- The order of the arguments of annotations may now be relevant, and in general arguments can thus no longer be freely re-ordered. Specific annotations may however allow the order to be irrelevant. For `@doc` annotations the argument order is relevant. For `@state` and

`@controller:properties` annotations the argument order is not relevant (issue #1192).

CIF text editor

- The CIF text editor now has greatly improved performance when the CIF model has many warnings or errors. This may in certain cases also prevent the IDE from freezing indefinitely (issue #995).

CIF type checker

- The CIF type checker now warns about discrete variables that are effectively constant, if this can be detected statically (issue #1163).
- The CIF type checker no longer crashes when checking certain annotations in invalid models (issue #1175).

CIF data-based synthesis

- Synthesis now supports CIF models with discrete and input variables that have integer types that allow negative integer values (issue #458).
- Synthesis now supports CIF models with discrete and input variables that have integer types that include value 2,147,483,647 as possible value. Instead, now the total number of possible values of the integer type must not exceed 2,147,483,647 (issue #458).
- Synthesis now supports CIF models with expressions that contain negative integer literals and unary `-` operations (issue #458).
- Synthesis now supports CIF models with expressions that contain binary `-` operations, regardless of where the binary subtractions are used, as long as the arguments of the binary subtractions are supported expressions as well. This removes the restriction that binary subtractions were only supported directly in the right-hand sides of assignments (issue #458).
- Synthesis now supports CIF models with expressions that use the `abs`, `sign`, `max` or `min` standard library functions on supported arguments (issue #1216).
- Synthesis now has improved performance in certain cases for CIF models with `-` (binary), `+` (binary), `div`, or `mod` operations (issue #458).
- Synthesis now has additional caching, to prevent duplicate work, and in certain situations even prevent exponential blow-ups. This concerns: the result of the pre-condition check for supported expression forms, for references to algebraic variables; the 'plants refer to requirements check' result for references to constants, discrete variables, continuous variables and algebraic variables; and the conversion result for converting references to algebraic variables in predicates and other expressions to the internal BDD representation (issue #1187).
- Synthesis now more efficiently creates BDD predicates for variables with many possible values. This concerns initial value predicates for discrete variables initialized using `in any`, update relation predicates for input variables, and implicit variable-never-out-of-range requirements (issue #1199).
- Synthesis now uses less memory when converting the synthesis result back to a CIF specification, in case the `nodes` BDD output mode is used (issue #1227).
- Removed some unnecessary computations related to BDD variable pairings, which may make synthesis faster, although it is unlikely this will be noticeable in practice (issue #1178).
- The advanced variable ordering option has a new variable orderer, the repeat orderer, that

allows to repeatedly apply a given orderer a number of times (issue #1165).

- The custom variable ordering (for both the basic and advanced options) no longer crashes or produces invalid results for name patterns with characters that have a special meaning in Java regular expressions (issue #1176).
- The custom edge ordering no longer crashes or produces invalid results for name patterns with characters that have a special meaning in Java regular expressions (issue #1185).
- Fixed the error message that is shown if an invalid custom BDD variable order is used (issue #1176).
- Fixed the debug output of the saturation exploration strategy for reachability computations, for variables in the edges/variables matrix, in case of custom variable orders where different CIF variables are interleaved (issue #1205).
- The documentation about supported specifications has been reduced and simplified (issue #458).
- Fixed a small issue in the advanced variable ordering documentation, for the reverse orderer (issue #1165).

CIF event-based toolset

- The event-based toolset documentation has some small textual improvements (issue #1229).

CIF multi-level splitter

- The CIF toolset now includes a new tool, the CIF multi-level splitter, which enables a form of non-monolithic synthesis. The multi-level splitter can be used to split a CIF specification into a multi-level tree of smaller co-operating partial specifications. The partial specifications can be synthesized individually, which is typically more efficiently than synthesizing the entire original specification. The resulting supervisors, synthesized from the partial specifications, can then together be used to control the system. See the tool's documentation for more information (issue #318).

CIF to mCRL2 transformer

- The tool now supports CIF models with multiple initial states. That is, it supports discrete variables with multiple initial values as well as automata with multiple initial locations. See the tool's documentation for more information (issue #1220).
- The tool now supports CIF models with input variables (issue #1220).
- The tool now supports CIF models with arbitrary initialization predicates in locations, so also those that are not statically-evaluable. It now also supports CIF models with initialization predicates in components, so outside locations. And it supports models without an initial state. See the tool's documentation for more information (issue #1220).
- The tool now supports CIF models with state/event exclusion invariants (issue #1220).
- The tool now supports CIF models with state invariants that do not depend on input variables (issue #1220).
- The tool now generates mCRL2 files with improved header comments and improved comments for value actions (issue #1220).
- The tool now uses the proper tool name in precondition violation reports (issue #1220).
- The tool no longer performs the *Add default initial values* CIF to CIF transformation as pre-

processing (issue #1220).

- The description of the tool's *Generate 'value' actions* option has been fixed (issue #1220).
- The tool's documentation has some small improvements and fixes (issues #1182 and #1220).

CIF controller properties checker

- The tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with discrete and input variables that have integer types that allow negative integer values (issue #458).
- The tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with discrete and input variables that have integer types that include value 2,147,483,647 as possible value. Instead, now the total number of possible values of the integer type must not exceed 2,147,483,647 (issue #458).
- The tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with expressions that contain negative integer literals and unary `-` operations (issue #458).
- The tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with expressions that contain binary `-` operations, regardless of where the binary subtractions are used, as long as the arguments of the binary subtractions are supported expressions as well. This removes the restriction that binary subtractions were only supported directly in the right-hand sides of assignments (issue #458).
- the tool now supports, for the bounded response, confluence and non-blocking under control checks, CIF models with expressions that use the `abs`, `sign`, `max` or `min` standard library functions on supported arguments (issue #1216).
- The tool now has, for the bounded response, confluence and non-blocking under control checks, improved performance in certain cases for CIF models with `-` (binary), `+` (binary), `div`, or `mod` operations (issue #458).
- The tool now has, for the bounded response, confluence and non-blocking under control checks, additional caching, to prevent duplicate work, and in certain situations even prevent exponential blow-ups. This concerns: the result of the pre-condition check for supported expression forms, for references to algebraic variables; and the conversion result for converting references to algebraic variables in predicates and other expressions to the internal BDD representation (issue #1187).
- The tool now more efficiently creates BDD predicates for variables with many possible values. This concerns initial value predicates for discrete variables initialized using `in any`, and update relation predicates for input variables (issue #1199).
- The bounded response check no longer crashes in certain cases where bounded response doesn't hold and there are very many possible states where there is no bounded response. There was already a limit on the number of states to print, but the calculation could result in integer overflow, which could lead to crashes. Some of such cases where already fixed in release v7.0, and the remaining cases have now been fixed as well (issue #1225).
- The *Maximum number of printed cycle states* option now accepts 2,147,483,639 as the largest number of cycle states to print, instead of 2,147,483,637 (issue #1225).

CIF PLC code generator (stable)

- Real literal values in generated code now have no more precision than is supported by the particular target for which the code is generated. This may prevent warnings when the generated code is imported into PLC environments. As a result of this change, real literals may now be generated in a different syntax, even if those literals were already within the precision of the target (issue #1146).

CIF documentation

- The CIF history page's ESCET papers section now includes an extra paper reference (issue #1230).

6.7. Version 7.0 (2025-03-31)

CIF language

- The CIF language now features input parameters, a new kind of parameters for component definitions (automaton and group definitions). For such parameters, only input variables and input parameters can be used as arguments in component instantiations. A primary use case for using input parameters instead of algebraic parameters, is that input parameters can be assigned by SVG input mappings within component definitions. See for more information the CIF tutorial lesson on automaton definition parameters, which has been extended to include input parameters (issue #1118).
- SVG input mappings with updates are no longer an experimental language feature. It is now considered a stable language feature (issue #1085).
- Using a `min` or `max` function with integer arguments now results in a ranged `int` value if *at least one* argument has a ranged `int` type. Previously, they *both* had to have a range for the result to be ranged. This for instance allows to convert the value of a variable `x` with a rangeless `int` type to a ranged `int` value of type `int[0..1000]` using `max(0, min(x, 1000))` (issue #1129).

CIF simulator

- The plot visualizer no longer crashes if the simulator is started as a command line tool and the plot visualizer window is resized, not even when the window is resized to an empty size. Note that this does not affect the CIF simulator plot visualizer in the ESCET IDE; not even if the plot visualizer is put into a separate window (issue #1142).
- The simulator documentation now documents the default value of the simulator's complete mode option (issue #1133).
- The simulator documentation now indicates that, with complete mode enabled, there are situations where not all possible transitions are printed (issue #1134).

CIF explorer

- The CIF explorer has various improvements and fixes relating to the precondition that there may not be too many initial states. These changes primarily prevent crashes and incorrect results. They may also lead to better estimates of the possible number of initial states, as well as improved output in case of a precondition violation. Performance of computing the initial states has also been improved in certain cases, but the tool could now give less feedback in case there is no initial state (issue #1110).

CIF data-based synthesis

- The tool may internally produce simpler predicates, such as `true or X` now becoming `true`, in the conversion of the specification to BDDs. This may lead to differences in conversion performance, variable ordering, and debug output. The benchmarks shown only minimal changes (less than 0.1% change in memory/time). However, for the `wafer_scanner_n1` benchmark, synthesis requires about 12% less BDD operations (issue #1154).
- Models with discrete variables or input variables with an integer type with a range that includes value 2,147,483,647 (the maximum integer value) are no longer supported, preventing crashes (issue #903).
- Debug output for models with large integer values is now proper, as it for instance no longer shows overflown values, and has improved readability by including thousand separators (issue #903).
- Debug output for variable ordering now shows BDD variable indices in the variable order overview tables (issue #903).

CIF controller properties checker

- The tool may internally produce simpler predicates, such as `true or X` now becoming `true`, in the conversion of the specification to BDDs. This may lead to differences in conversion performance, variable ordering, and debug output (issue #1154).
- Models with discrete variables or input variables with an integer type with a range that includes value 2,147,483,647 (the maximum integer value) are no longer supported, preventing crashes (issue #903).
- Debug output for models with large integer values is now proper, as it for instance no longer shows overflown values, and has improved readability by including thousand separators (issue #903).
- Debug output for variable ordering now shows BDD variable indices in the variable order overview tables (issue #903).
- The bounded response check no longer crashes when bounded response doesn't hold and there are too many possible states where there is no bounded response. There was already a limit on the number of states to print, but the calculation could result in integer overflow, which could lead to crashes. This has been fixed (issue #1138).
- The tool is now more responsive to termination requests (issue #996).

CIF code generator

- The generator now has an *HTML frequencies* option to configure the minimum, default and maximum frequency for generated HTML pages. The frequencies can be changed, to for instance ensure that the execution is fluent and accurate enough. See the documentation of the option for further information (issue #1127).
- Generated HTML pages now properly handle SVG output mappings for all attributes, regardless of whether the attributes are supported as SVG presentation attributes and/or CSS style properties, regardless of which of those takes precedence, and regardless of whether they are already present on the element or not (issue #715).
- Generated HTML pages now properly handle SVG output mappings where the value is not a valid CSS value for the corresponding CSS style property, ensuring that the last-set valid

value is not preserved (issue #1157).

- Fixed a typo in the HTML code generation documentation (issue #1127).

CIF PLC code generator (stable)

- The generator now supports models that use the `ceil`, `floor`, `round`, `scale` and `sign` functions (issue #1109).
- The ESCET IDE now ships with CIF PLC code generator 'self-certification' tests. There is an extra option on the *Welcome* screen to import them. The import wizard can also be accessed via **File › New › Example**. These 'self-certification' tests can be used to test the PLC code generator in external PLC development environments. See also the tool's documentation for more information (issues #1104 and #1109).
- Fixed a syntax highlighting issue in the code generator's documentation (issue #1147).

CIF to CIF transformer

- A new *Eliminate state invariants* CIF to CIF transformation has been added. It comes in three variants: one that eliminates all state invariants, one that only eliminates plant state invariants, and one that only eliminates supervisor state invariants. For more information, see the documentation of the new transformation (issue #1087).
- The *Eliminate 'if' updates* CIF to CIF transformation no longer supports CIF specifications with component definitions/instantiations. Eliminate component definitions/instantiations using the corresponding CIF to CIF transformation before applying the *Eliminate 'if' updates* transformation (issue #1118).
- The *Anonymize names* CIF to CIF transformation now uses the `iparam` prefix when renaming input parameters (issue #1118).
- Fixed some typos in the CIF to CIF transformer documentation (issues #1106 and #1118).
- The *Linearize (merge)* and *Linearize (product)* transformations may produce simpler linearized predicates, such as `true or X` now becoming `true`, for edge guards, urgency guards, and initialization/marker predicates from locations (issue #1154).
- The *Eliminate monitors* transformation may produce simpler predicates, such as `true or X` now becoming `true`, for edge guards and send values (issue #1154).

CIF to Supremica transformer

- The transformer may produce simpler Supremica predicates, such as `1 | X` now becoming `1`, for edge guards and state invariant predicates (issue #1154).

CIF to UPPAAL transformer

- The transformer may produce simpler UPPAAL predicates, such as `true || X` now becoming `true`, for edge guards (issue #1154).

CIF website

- The CIF website now shows the CIF logo (issue #1140).

CIF documentation

- The CIF reference manual has small textual improvements (issue #1071).
- The *Developers* section of the documentation now contains a page about how to reliably extract information from a CIF specification, by converting the CIF specification to XML and using XML queries (issue #1071).

6.8. Version 6.0 (2024-12-20)

CIF type checker

- Warnings are now added for SVG output mappings that overwrite (rather than set) the `transform` attribute of an SVG element (issue #1021).
- Warnings are now added for SVG output mappings with a transformation on an element whose ancestor has a `matrix`, `rotate`, `scale` or `skew` transformation or a variant of them (issue #1021).
- Warnings are now added for all convoluted references to addressables in updates of edges of automata, regardless of whether the reference is via a component instantiation or component parameter or not (issue #227).

CIF text editor

- The dark theme now uses a lighter red color for string literals, for better contrast (issue #966).

CIF simulator

- The simulator now supports SVG input mappings with updates to input variables (issues #227 and #778).
- The simulator now has an *Execution mode* option, to enable execution mode. Execution mode is disabled by default. With execution mode, the simulator adheres to the execution scheme defined by the CIF controller properties checker. See the documentation of the new option for further information (issue #1045).
- The simulator now has a new *Non-urgent events* option that replaces the *Environment events* option, which is no longer supported (issue #1063).
- The simulator can no longer execute external user-defined functions asynchronously; it now always executes them synchronously. This means that termination requests are no longer processed during the execution of external user-defined functions. The simulator's *External functions asynchronous execution* option is no longer supported (issue #656).
- The simulator runtime error messages for variables getting assigned out of bounds values now use absolute variable names for clarity (issue #227).
- The simulator documentation has some small improvements and fixes (issues #1063 and #1090).

CIF data-based synthesis

- When using the saturation exploration strategy for reachability computations, the debug output now includes an edges/variables matrix that gives insights into the complexity of the model (issue #957).
- A BDD variable leak in computing edge support variable sets has been fixed, which may slightly improve synthesis performance (issue #1006).

CIF controller properties checker

- The confluence check's performance has been greatly improved, by interleaving the internally added 'zero' BDD variables (issue #1006).
- The confluence check is now more responsive to termination requests (issue #1008).

- The confluence check now produces improved debug output in case the model has exactly one controllable event (issue #1007).
- The bounded response check now prints the edges that are part of cycles, in case bounded response doesn't hold due to cycles (issue #621).
- The bounded response check now prints states that are part of cycles, in case bounded response doesn't hold due to cycles. By default 10 states are printed, but this can be configured through the new *Maximum number of cycle states* option (issue #621).
- The bounded response check's debug output has been slightly improved (issue #621).
- The non-blocking under control check's debug output now includes an edges/variables matrix that gives insights into the complexity of the model (issue #957).
- The performance of the bounded response, confluence and non-blocking under control checks may have slightly improved, due to fixing a BDD variable leak in computing edge support variable sets (issue #1006).
- The order of the checker's options has changed. It now first lists all enablement options, before their more detailed configuration. This now matches the order in the documentation (issue #621).
- The documentation now explains for each check that prints additional details what details are printed (issue #621).
- The documentation has some small fixes (issue #1045).

CIF code generator

- The generator now by default generates HTML, not Simulink code (issue #715).
- The generator now automatically adapts the default code prefix to be a valid prefix, preventing having to specify a custom code prefix. If the user specifies a custom code prefix, it is not automatically adapted and thus still a valid CIF identifier must be specified manually (issue #1084).
- The generator for HTML now supports SVG input mappings with updates to input variables (issue #227).
- The generator for HTML now produces code that initializes input variables to their default values (issue #227).
- The generator for HTML now produces code that includes input variables and their values when printing states to the log panel (issue #227).
- The generator for HTML now produces code that properly executes SVG input events before uncontrollable events (issue #227).
- The generator for HTML now produces code that processes all SVG image interactive element clicks in a single execution of the corresponding event loop, even if certain SVG input events are not enabled, thus ensuring that all clicks have been processed before executing the uncontrollable events event loop (issue #227).
- The generator for HTML now produces code that logs runtime errors to the log panel (issue #715).
- The generator for HTML now produces code that results in a runtime error if an SVG input mapping does not map to an event at runtime (issue #1069).

- The generator for HTML now produces code with nicer layout for SVG input mappings with event mappings (issue #1069).
- The generator's documentation has some textual improvements (issues #227 and #1045).
- The generator for JavaScript and HTML now wraps runtime errors in more context, making them easier to understand (issue #715).
- The generator for JavaScript and HTML now generates code with improved type hint consistency in generated JSDocs (issue #715).
- The generator for Java now takes the new keywords of Java 21 into account (issue #656).
- The generator now properly handles type declarations, by eliminating them as preprocessing. This among others fixes a bug for generation of out of range checks for variables with as type a type declaration with a ranged integer type (issue #998).
- The generator now properly handles tuple field projections in case of compatible tuple types (issue #998).

CIF PLC code generator (deprecated)

- The CIF PLC code generator that was previously labeled as the 'stable' version, is now the 'deprecated' version (issue #1092).

CIF PLC code generator (stable)

- The CIF PLC code generator that was previously labeled as the 'experimental' version, is now the 'stable' version. Support for some targets is still experimental. See the tool's documentation for further information (issue #1092).
- The new *Input/output code form* option can be used to generate functions for reading inputs and writing outputs, rather than having the code be generated in the main program (issue #1055).
- The *Event transitions form* option now has an extra possibility. Use *Event code is combined into a function by event controllability and CIF scope* to obtain event functions for all (un)controllable events in each scope (issue #1031).
- The generator now writes, for the S7 target, a text file `derived_type_dependencies.txt`. The file contains the dependency information between structs, and from structs to enumerations if the latter types are used in a struct. The information from this file can be used to determine the order in which to create program blocks from the various data types (issue #944).
- The `loopsKilled` variable in the generated code is now named `loopsExhausted`. Its existence and objective are now documented in the generator's documentation (issue #1013).
- The model overview has been moved to the very end of the body of the generated main program. Its title has also been improved (issue #1022).
- The generated code now has improved documentation and comments for PLC internal variables, i.e., current-location variables and edge selection variables for automata (issue #1023).
- Due to internal changes in the generator, the order of the declarations in the generated code may be different (issue #993).
- The generator no longer crashes on models with nested arrays. Note that nested arrays were already unsupported, as indicated in the documentation (issues #1017 and #1025).

- The generator now correctly handles compatible enumerations if enumerations are kept in the generated code (issue #1018).
- The generator now provides improved error messages when reading invalid CSV files, in general, as well as specifically when not all rows have the same number of fields (issue #1044).
- Fixed a typo in the command line description of the generator's *Simplify values* option (issue #1055).
- Improved the command line and GUI option descriptions of the generator's *PLC maximum iterations* option (issue #1055).
- For the TwinCAT target, the generator now properly considers (in)equality and comparison operators to have the same priority, where applicable forcing the use of parentheses in the generated code (issue #1040).
- For the TwinCAT target, the generated code now properly uses `VAR` for temporary variables in functions, rather than `VAR_TEMP` (issue #1038).
- For non-S7 targets, if any code is generated in functions rather than in the main program, state variables are now stored in a global variable list instead of in the main program (issue #1039).
- For non-S7 targets, the generated code now correctly ensures short-circuit evaluation semantics for the `=>`, `and` and `or` binary operators (issue #1037).
- For non-S7 targets, the generated code no longer has type extensions for the `SEL` function (issues #1036 and #1041).
- For the S7 targets, `any_derived` is now considered a keyword, rather than the misspelled `and_derived` keyword (issue #679).
- For the S7-1200/1500 targets, the generator now considers the same naming restrictions as for the S7-300/400 targets (issue #1061).
- The documentation for the S7 targets now lists a potential fix for the *System Fault* PLC error (issue #1034).
- Various documentation improvements (issues #1044, #1045 and #1055).

CIF merger

- The documentation has been slightly improved (issue #1045).

CIF to CIF transformer

- A new *Convert to interface* transformation is now available that converts a CIF specification to its interface. It comes in two flavors, one to generate a full interface and one to generate a reduced interface (issues #1067 and #1086).
- The *Anonymize names*, *Eliminate groups* and *Lift events* transformations now remove controller properties annotations (issue #1001).
- The *Linearize (merge)* transformation no longer removes controller properties annotations (issue #1001).
- The documentation of the *Linearize* transformations has been improved (issue #1045).

CIF to Supremica transformer

- Fixed a typo in the command line description of the tool's *Eliminate enumerations* option (issue #1066).

CIF examples

- Fixed a duplicate section name in the *bridge* example readme (issue #1081).

CIF documentation

- The CIF/SVG visualization and interaction documentation that is not specific to the simulator has been moved from the CIF simulator documentation to the CIF language tutorial and CIF language reference manual (issue #1015).
- The print declarations and text formatting documentation that is not specific to the simulator has been moved from the CIF simulator documentation to the CIF language tutorial and CIF language reference manual (issue #1048).
- The CIF tutorial lesson on input variables now has an example of using input variables with SVG interaction, by means of SVG input mappings with updates (issue #227).
- The CIF tutorial lesson on imports has been slightly improved (issue #1045).
- The CIF documentation on the ESCET website and in the Eclipse help now use a darker green color for numbers in source code, for better contrast (issue #1030).

6.9. Version 5.0 (2024-10-03)

Language changes:

- Annotations are now a stable CIF language feature (issue #869).
- The semantics of the uncontrollable/controllable events bounds of the controller properties annotation have changed. They no longer represent bounds on the lengths of sequences of uncontrollable/controllable events. Instead, they now represent bounds on the number of iterations of the uncontrollable/controllable event loops, under the assumption of the execution scheme prescribed by the controller properties checker (issue #971).

Deprecations and removals:

- The CIF to mCRL2 transformer has been re-implemented and is now based on first linearizing the CIF specification. It generates mCRL2 models with a single process. Therefore, the options related to specifying and debugging instance trees have been removed. For further details, see below (issue #352).
- The CIF to mCRL2 transformer's *Generate value actions* option has been renamed from `--read-values`/`-r` to `--value-actions`/`-v` (issue #352).
- The CIF data-based synthesis tool's 'Edge workset algorithm' option is no longer supported, and is replaced by the new 'Exploration strategy' option. This new option is also described below (issue #947).

Improvements and fixes:

- The CIF type checker now reports component definition parameter count versus instantiation

argument count mismatches on the component definition reference, rather than on the instantiation name (issue #973).

- Various CIF tools have improved termination checking performance (issue #936).
- CIF tools that fail to save a CIF XML file now give the correct error message (issue #881).
- CIF tools that write CIF files have improved performance, especially when using Eclipse platform paths (issue #42).
- The CIF simulator has improved output in case of precondition violations (issue #424).
- The CIF to yEd transformer has improved output in case of precondition violations (issue #424).
- The CIF to CIF transformer's linearization transformations have an improved unsupported message in case linearization is applied to a CIF specification without automata (issue #944).
- The CIF to CIF transformer's linearization transformations now produces models with the elements of the specification in a certain specific order, for compatibility with the execution scheme prescribed by the CIF controller properties checker (issue #935).
- The CIF to CIF transformer's linearize merge transformation now removes the controller properties annotation from the input specification, if present (issue #896).
- The CIF to CIF transformer's convert (un)controllable events to (un)controllable transformations now remove the controller properties annotation from the input specification, if present (issue #896).
- The CIF to CIF transformer's remove requirements transformation now removes the controller properties annotation from the input specification, if present (issue #896).
- The CIF explorer has improved output in case of precondition violations (issue #424).
- The CIF explorer precondition check for not having too many potential initial states has been improved (issue #424).
- The CIF merger now removes the controller properties annotation from the input specification, if present (issue #896).
- The CIF data-based synthesis tool now has a new 'Exploration strategy' option, featuring not only regular fixed edge order and the workset algorithm, but also a new third strategy, the saturation strategy. This new saturation strategy improves the performance of reachability computations. On average, for the CIF benchmark models, synthesis performance is improved 14.5 times, although the results differ per model. Saturation is now the new default exploration strategy. The 'Edge workset algorithm' option is no longer supported (issues #947, #957, #958 and #992).
- The CIF data-based synthesis tool now has improved debug and warning output (issues #938 and #953).
- The CIF data-based synthesis tool now has improved numbers for the number of states in the controlled system, and the number of true paths in the debug output of BDDs for large predicates. It indicates more precise numbers for larger BDDs, and no longer indicates wrong numbers for very large BDDs. This comes at the cost of computing these numbers taking longer, and the cache for computing these numbers requiring more memory (issue #951).
- The CIF data-based synthesis tool no longer crashes when a termination request is given while converting updates to BDDs (issue #909).

- The CIF data-based synthesis tool has improved output in case of precondition violations (issue #424).
- The CIF data-based synthesis tool now supports CIF specifications with predicates that are not one of the supported forms, but that can be statically evaluated to a single value without giving evaluation errors (issue #424).
- The CIF data-based synthesis tool's 'BDD debug max paths' option now supports any non-negative integer value (however large it is). The value may no longer be entered as a real value, e.g., no more '1e38' (issue #951).
- The CIF data-based synthesis tool now removes the controller properties annotation from the input specification, if present (issue #896).
- The CIF data-based synthesis tool has a fix for an integer overflow bug, which may trigger while checking for the need to resize the node array, causing large node arrays to grow unnecessary (issue #947).
- The CIF controller properties checker now uses the saturation strategy for reachability computations, improving the performance of the non-blocking under control check (issues #947, #957 and #958).
- The CIF controller properties checker now computes confluence internally using BDDs rather than MDDs, for better performance (issues #693, #695 and #997).
- The CIF controller properties checker now computes bounded response differently, using the execution scheme, which improves the performance of the check, and generally leads to lower bounds (issue #971).
- The CIF controller properties checker no longer supports non-determinism for controllable events, regardless of what checks are performed, as was already documented (issue #695).
- The CIF controller properties checker now converts the input CIF specification to an internal representation before each check, rather than at the start. This may reduce the required memory if multiple internal representations are required (issue #971).
- The CIF controller properties checker now warns if the input specification has no input variables, indicating that the hardware mapping is missing (issue #977).
- The CIF controller properties checker has improved output in case of precondition violations (issue #424).
- The CIF controller properties checker has improved debug output (issues #695, #937 and #938).
- The CIF controller properties checker now has improved numbers for the number of true paths in the debug output of BDDs for large predicates. It indicates more precise numbers for larger BDDs, and no longer indicates wrong numbers for very large BDDs. This comes at the cost of computing these numbers taking longer, and the cache for computing these numbers requiring more memory (issue #951).
- The CIF controller properties checker no longer has indented empty lines in its console output (issue #406).
- The CIF controller properties checker no longer crashes when a termination request is given, and is more responsive to termination requests (issues #909 and #964).
- The CIF controller properties checker for bounded response and non-blocking under control,

now supports CIF specifications with predicates that are not one of the supported forms, but that can be statically evaluated to a single value without giving evaluation errors (issue #424).

- The CIF controller properties checker for bounded response and non-blocking under control, has a fix for an integer overflow bug, which may trigger while checking for the need to resize the node array, causing large node arrays to grow unnecessary (issue #947).
- The CIF controller properties checker documentation now documents the prescribed execution scheme (issue #970).
- The CIF code generator no longer supports specifications with events that are not controllable or uncontrollable (issue #628).
- The CIF code generator now generates code that adheres to the execution scheme prescribed by the CIF controller properties checker. As a result of this, some parts of the generated code are now in a different order. Also, the generated code now has separate event loops for SVG input events (HTML output only), uncontrollable events, and controllable events. Furthermore, if in an event loop a transition for an event is executed, the iteration now continues with the next event, rather than starting at the first event in a next iteration. Finally, in generated C code, the configurable value `MAX_NUM_EVENTS` has been renamed to `MAX_NUM_ITERS` (issues #628 and #970).
- The CIF code generator now generates code that avoids ambiguity between event and edge indices (issue #952).
- The CIF code generator now generates code with improved comments (issue #952).
- The (still experimental) new CIF PLC code generator now allows to custom I/O variable names in the I/O table file, using an optional fourth column (issue #897).
- The (still experimental) new CIF PLC code generator now allows algebraic variables to be used as outputs in the I/O table file (issue #956).
- The (still experimental) new CIF PLC code generator will now by default, through its 'PLC maximum iterations' option, compute the iteration limits from the bounds captured in the bounded response information of the input specification's controller properties annotation, if available. It still has a fallback to infinite loop bounds, and the limits can still be specified manually (issues #816 and #965).
- The (still experimental) new CIF PLC code generator now supports documentation annotations of various CIF objects and includes them in the generated PLC code (issue #814).
- The (still experimental) new CIF PLC code generator now allows differently sized types for the I/O port and the corresponding CIF variable (issue #884).
- The (still experimental) new CIF PLC code generator now generates code that adheres to the execution scheme prescribed by the CIF controller properties checker. As a result of this, some parts of the generated code may be in a different order (issues #935 and #970).
- The (still experimental) new CIF PLC code generator now warns about generating PLC code for a specification that was not checked using the CIF controller checker, or if it was checked but some of the checks didn't hold (issues #816 and #965).
- The (still experimental) new CIF PLC code generator now allows generating event transition code in separate functions. That is, it has a new 'Event transitions form' option, to allow generating all event code in the `MAIN` program (still the default), or in a function per event (avoids issues if code becomes large). Due to these changes, some generated code may now be in

a different order in the generated PLC code (issues #912, #935, #980 and #986).

- The (still experimental) new CIF PLC code generator has various improvements for generated edge selection variables, such as not generating one for automata that don't need it, improved naming, using smaller data types if possible, and starting at zero rather than one to more often allow using smaller data types (issue #923).
- The (still experimental) new CIF PLC code generator has an improved option description for the 'PLC maximum iterations' option (issue #965).
- The (still experimental) new CIF PLC code generator has improved error messages in case invalid values are provided for the 'PLC maximum iterations' option (issue #965).
- The (still experimental) new CIF PLC code generator has improved handling of enumerations (including those generated for locations of automata), using bit-like types rather than integers, using smaller types when possible, and having more direct traceability to the input CIF model (issue #934).
- The (still experimental) new CIF PLC code generator has improved comments for applying updates of monitor automata (issue #814).
- The (still experimental) new CIF PLC code generator has improved layout for the model overview in the generated code (issue #814).
- The (still experimental) new CIF PLC code generator has improved names for generated PLC tuple types, in general, but especially for CIF tuple types that are part of a CIF type declaration (issue #679).
- The (still experimental) new CIF PLC code generator has improved names in the generated code (issue #945).
- The (still experimental) new CIF PLC code generator has improved loop killed detection in the generated code (issue #965).
- The (still experimental) new CIF PLC code generator now ensures that generated S7 temporary variable tables have the required minimum size (issue #860).
- The (still experimental) new CIF PLC code generator now avoids generating parentheses around real literal values for variable and constant declarations (issue #877).
- The (still experimental) new CIF PLC code generator now checks that only boolean, integer and real-typed discrete and input variables in a CIF specification are used for I/O, regardless of whether explicit PLC types are given in the I/O table file (issue #884).
- The (still experimental) new CIF PLC code generator no longer crashes on reading an invalid I/O table file, but instead gives a proper error message (issue #967).
- The (still experimental) new CIF PLC code generator now generates correct code for constants having the value of another constant as its value, if enumeration literals are eliminated to constants (issue #969).
- The (still experimental) new CIF PLC code generator now generates code based on the correct operator priority for the binary subtraction operator (issue #968).
- The (still experimental) new CIF PLC code generator for S7-300 and S7-400 now handles additional keywords, reserved identifiers, and so on (issues #901 and #921).
- The (still experimental) new CIF PLC code generator now has improved error messages when

reading invalid I/O table files (issue #967).

- The (still experimental) new CIF PLC code generator rename warnings for location pointers no longer print a spurious period at the end of the location pointer names (issue #984).
- The (still experimental) new CIF PLC code generator check whether an I/O address exceeds the size of the largest compatible type supported by the target, now correctly considers the different possible I/O types, rather than always checking for integer types (issue #884).
- The (still experimental) new CIF PLC code generator documentation for S7 now explains how to ensure that properly-compiling program blocks can be generated in TIA Portal for PLC struct types generated for CIF tuple types (issue #863).
- The CIF to mCRL2 transformer has been re-implemented and is now based on first linearizing the CIF specification. For recursive process call, now only the assigned variables are included, not all variables, and the arguments are named rather than positional, which reduces the model size and improves readability. No more summations, nor location pointers (sorts) for automata with only one location, are generated (issues #352, #950, #954 and #955).
- The CIF to mCRL2 transformer no longer supports specifications with urgent locations and edges (issue #352).
- The CIF to mCRL2 transformer now supports specifications with `tau` events, the `<=>` binary operator, the `div` and `mod` binary operators on positive divisors, cast expressions that don't change the type, location references, channels and receive expressions, initialization predicates in components that are trivially `true`, non-restrictive invariants, `if` expressions, `switch` expressions, algebraic variables with equations, and `if` updates (issues #226, #229 and #948).
- The CIF to mCRL2 transformer now generates mCRL2 specifications with a `marked` action. This action can be used in properties for verification. The generation of the action is enabled by default, but can be disabled using the new 'Generate marked action' option (issue #353).
- The CIF to mCRL2 transformer now uses a different naming scheme, based on absolute names. See the tool's documentation for details (issues #352 and #224).
- The CIF to mCRL2 transformer now generates comments in the generated mCRL2 models (issue #352).
- The CIF to mCRL2 transformer now performs less preprocessing before the precondition check, for improved traceability of precondition violations.
- The CIF to mCRL2 transformer no longer eliminates automaton `self` references as preprocessing, and additionally eliminates type declarations and performs linearization as preprocessing (issue #352).
- The CIF to mCRL2 transformer has improved output in case of precondition violations (issue #424).
- The CIF to mCRL2 transformer has improved documentation (issues #352 and #424).
- The CIF event-based tools have improved output in case of precondition violations (issue #424).
- The CIF event-based tools no longer support unused channels (issue #424).
- The CIF event-based tools now support non-restrictive invariants in components, non-restrictive state/event exclusion invariants in locations, multiple non-restrictive state invariants per location, and any statically-determinable non-restrictive state invariant rather than only literal

`true` (issue #424).

- The CIF event-based tools now support having multiple initialization/marker predicates in a location, and any initialization/marker predicate in locations that can be statically evaluated rather than only boolean literals (issue #424).
- The CIF event-based tools now support having multiple guards on an edge of a location, and any guard predicate that can be statically evaluated rather than only boolean literals (issue #424).
- The CIF event-based controllability check no longer supports unused events that are not controllable or uncontrollable (issue #424).
- The CIF event-based supervisory synthesis tool no longer supports unused events that are not controllable or uncontrollable (issue #424).
- The CIF event-based supervisory synthesis tool no longer checks for no marking during precondition checking, but now does so as a first step during synthesis. Having no marking still results in an empty supervisor (issue #424).
- The CIF event-based language equivalence check no longer has indented empty lines in its console output (issue #406).
- The CIF event-based DFA minimization tool no longer has indented empty lines in its console output (issue #406).
- The CIF event disabler now supports component definitions/instantiations and eliminates them as preprocessing (issue #424).
- The CIF event disabler now removes the controller properties annotation from the input specification, if present (issue #896).
- The CIF benchmarks overview generation script now consistently generates `?` instead of `-1` (issue #906).
- The CIF benchmarks have improved and newly-added model description (issue #949).
- The CIF tutorial lesson on initialization predicates has been improved (issue #739).
- Various documentation improvements and fixes (issues #424, #628, #742, #816, #849, #863, #929, #931, #943, #946, #956, #965, #970, #971, #976 and #987).

6.10. Version 4.0 (2024-06-30)

Language changes:

- The syntax of annotation arguments has changed, to allow annotation arguments without a name. Each annotation has its own requirements, for instance requiring names for all its arguments, requiring all its arguments to not have a name, requiring names only for some of its arguments, or even allowing arguments to optionally have a name. If both the argument name and value are present, a colon (`:`) should now be used between them rather than an equal sign (`=`). Annotations are still an experimental work-in-progress language feature. Their design may change in a backward incompatible manner (issue #801).
- The `@doc` annotation now requires one or more unnamed arguments, rather than a single argument named `text`. See the documentation about documentation annotations for further details (issue #801).

- The CIF language now features a new `controller:properties` annotation. It is used by the CIF controller properties checker, to add the results of the performed checks to the input model (issues #410 and #621).
- Annotations may now be added to algebraic parameters of component definitions, automata, automaton definitions, automaton instantiations, edges, enumerations, enumeration literals, events, functions, function parameters, function variables, groups, group definitions, group instantiations, invariants, specifications and type declarations. Annotations on specifications, invariants in locations, and edges require double `@`-signs, such as `@@doc` instead of `@doc` (issues #593 and #824).
- The CIF type checker now warns about certain declarations that have a type that supports only one value. This new check is enabled for algebraic parameters, algebraic variables, channel parameters, channels, discrete variables, function parameters, function variables, function return types and input variables. To resolve such a warning, check the type, or use a constant instead (issues #797 and #827).
- The CIF type checker now warns about state/event exclusion invariants with events that are not in the send, receive or synchronization alphabet of the specification. Such state/event exclusion invariants do not restrict any behavior (issue #838).
- The CIF type checker now has extended convoluted references detection. It now also detects certain convoluted references for types of component parameters (issue #856).

Deprecations and removals:

- The ball sorting model has been removed from the CIF benchmark models set (issue #745).

New features:

- The CIF to CIF transformer now has a new transformation to relabel supervisors as plants (issue #834).
- The CIF controller properties checker now has an additional check, the bounded response check. The bounded response check improves upon the finite response check, by checking for finite response also for uncontrollable events, not just for controllable events. Additionally, for both controllable and uncontrollable events, the new check also computes the bounds on the number of transitions that can be executed. Furthermore, the new check does not suffer from false negatives. The bounded response check is now recommended instead of the finite response check (issue #621).
- The CIF controller properties checker now has an additional check, the non-blocking under control check, that should hold for all supervisor models before controller code is generated from them. See the tool's documentation for further information (issue #410).

Improvements and fixes:

- The CIF parser is now more efficient in parsing elements that can have annotations, but do not have them (issue #593).
- The CIF pretty printer, which is used by all CIF tools to write output CIF models, now uses a separate `initial`, `marked` or `equation` keyword for each initialization predicate, marker predicate or equations, respectively. This also improves the traceability for certain precondition violations

being reported by tools that use the new precondition checking framework (issue #807).

- The CIF simulator has a fix to prevent resource leaks for specifications with external user-defined Java functions (issue #635).
- The CIF simulator now removes all annotations from the input model as pre-processing (issue #736).
- The CIF to CIF transformation to anonymize names now renames local variables of functions to `fvarNN` rather than to `discNN`, with `NN` a number (issue #811).
- The CIF to CIF transformation to eliminate state/event exclusion invariants to automata now creates a separate automaton for each named state/event exclusion invariant, to preserve their names as best as possible (issue #830).
- The CIF to CIF transformation to eliminate state/event exclusion invariants to automata now creates the automata for state/event exclusion invariants of groups within those groups, rather than besides those groups (issue #830).
- The CIF to CIF transformation to eliminate state/event exclusion invariants to automata now correctly handles `disables` state/event exclusion invariants of locations of automata with more than one location (issue #830).
- The CIF to CIF transformation to eliminate state/event exclusion invariants now removes state/event exclusion invariants for events that are not in the send, receive or synchronization alphabet of the specification, rather than converting them to automata. This prevents the events from becoming part of the synchronization alphabet of the specification, which would mean transitions could then become possible for these events, while that was not the case before the transformation. If any such invariants are encountered, the transformation now prints a warning (issue #838).
- The CIF to CIF transformation to transform `switch` expressions to `if` expressions now also transforms nested `switch` expressions (issue #858).
- The CIF to CIF transformer now checks the output model for annotation constraint violations (issues #823 and #853).
- The CIF merger now has improved output in case of annotation constraint violations detected on the merged model (issue #823).
- The CIF data-based synthesis tool has improved performance, by using compounded operations for applying edges, using partial transition relations, and taking runtime errors into account once before the main synthesis fixed point computations, rather than repeatedly during synthesis. On average it is about 3.4 times faster, and it also uses less memory. However, the gain depends on the model being synthesized (issues #865, #866, #867 and #878).
- The CIF data-based synthesis tool no longer crashes when the node table gets too big (issue #625).
- The CIF data-based synthesis tool now runs for a little bit longer before running out of BDD nodes array memory, as the nodes array is increased one more time to the maximum limit, before failing due to running out of memory (issue #625).
- The CIF data-based synthesis tool now allows converting resulting predicates explicitly to only CNF or only DNF, through configuration of its *BDD output mode* option (issue #873).
- The CIF data-based synthesis tool now detects and warns about more potential issues for events

that are disabled already before the main synthesis fixed point computations are started (issue #878).

- The CIF data-based synthesis tool now has improved warnings about state/event exclusion invariants with events that are not in the send, receive or synchronization alphabet of the specification (issue #838).
- The CIF data-based synthesis tool now has slightly improved debug output for applying plant invariants (issue #269).
- The CIF data-based synthesis tool now checks the output model for annotation constraint violations (issue #823).
- The CIF data-based synthesis tool now has workset algorithm dependency sets that take into account the potential for runtime errors to occur as edges are applied, which may reduce the number of dependencies and potentially improves synthesis performance if the workset algorithm is used (issue #878).
- The CIF explorer now uses proper terminology related to locations and edges versus states and transitions in its statistics output, debug output, report files, option descriptions and documentation (issues #691 and #882).
- The CIF explorer now has a new *Remove duplicate transitions* option that replaces the *Enable edge minimization* option. The latter is no longer supported. Functionally, the new option works as the old one (issue #691).
- The CIF explorer now removes all annotations from the input model as pre-processing (issue #736).
- The CIF code generator now supports documentation annotations for enumeration literals, events, functions, function parameters, function variables and specifications (issue #593).
- The CIF code generator for HTML now properly escapes SVG element identifiers when used in CSS selectors, making it possible to use SVG element identifiers with dots and colons (issue #715).
- The CIF code generator for C and Simulink now quote event names in generated comments (issue #593).
- The CIF code generator for Simulink now generates an empty line between the file header and imports (issue #593).
- The CIF code generator for Java, JavaScript and HTML now always uses proper paragraph syntax in the generated code for documentation from documentation annotations (issue #802).
- The CIF code generator for Java, JavaScript and HTML now generates improved comments in the generated code for user-defined functions (issue #593).
- The CIF code generator no longer crashes on reporting static evaluation errors for documentation annotations on nameless objects (issue #621).
- The CIF code generator now removes all non-`doc` annotations from the input model as pre-processing, and ignores annotations arguments during precondition checking (issue #736).
- The (still stable) current CIF PLC code generator now removes all annotations from the input model as pre-processing (issue #736).
- The (still experimental) new CIF PLC code generator now only supports events that are declared

as controllable or uncontrollable. This thus also excludes `tau` events, and edges without events that implicitly have a `tau` event (issue #800).

- The (still experimental) new CIF PLC code generator no longer supports the `cbrt` and `pow` standard library functions (issue #876).
- The (still experimental) new CIF PLC code generator no longer supports events that are only used as monitor events, and could thus be executed infinitely often during a PLC cycle (issue #679).
- The (still experimental) new CIF PLC code generator no longer supports arrays of size zero (issues #679 and #794).
- The (still experimental) new CIF PLC code generator no longer supports `switch` expressions on array or tuple-typed values, or parts of values (issue #859).
- The (still experimental) new CIF PLC code generator no longer crashes for specifications with nested tuples (issue #818).
- The (still experimental) new CIF PLC code generator no longer crashes for models with `switch` expressions (issue #857).
- The (still experimental) new CIF PLC code generator has some robustness improvements (issues #628 and #770).
- The (still experimental) new CIF PLC code generator now generates additional and improved comments, such as section comments for different parts of the generated code (issue #815).
- The (still experimental) new CIF PLC code generator now generates a comment with an overview of the generated code as it relates to the elements of the input CIF model (issue #815).
- The (still experimental) new CIF PLC code generator now generates a header comment in the generated code that lists the name and version of the code generator, the date/time of generation, and a brief explanation of how to interpret the generated code and how it links to the concepts of CIF. A new *Program header text file path* option has been added to the tool to customize/extend the generated header comment (issue #815).
- The (still experimental) new CIF PLC code generator now generates PLC code with two event loops, first one for the uncontrollable events and then one for the controllable events. The option to set a maximum on the number of iterations now accepts either one limit (for both loops) or two limits (separate limits for the two loops). And the tool now warns if the configured limit is reduced due to the target PLC model not supporting such large numbers (issue #628).
- The (still experimental) new CIF PLC code generator now treats `timer` as a reserved keyword of PLCs, to ensure proper code is generated for various targets (issue #679).
- The (still experimental) new CIF PLC code generator now explicitly documents that it reads the inputs at the start of the PLC program, and writes the outputs at the end of the PLC program (issue #831).
- The (still experimental) new CIF PLC code generator now generates proper code for models with double negations, such as `--5`. This affects the code generated for all unary expressions (issue #848).
- The (still experimental) new CIF PLC code generator now generates proper code for the projection on arrays through negative integer values (issues #854 and #862).

- The (still experimental) new CIF PLC code generator now generates proper code for the `min` and `max` standard library functions, for S7 targets (issue #847).
- The (still experimental) new CIF PLC code generator now generates proper code for models without any behavior (issue #860).
- The (still experimental) new CIF PLC code generator has various fixes related to handling timers (issues #770, #854 and #862).
- The (still experimental) new CIF PLC code generator no longer inlines `bool`, `int`, `real` and `enum` -typed constants, for IEC 61131-3, PLCopen XML, S7 and TwinCAT targets (issues #679 and #877).
- The (still experimental) new CIF PLC code generator now removes all non-`doc` annotations from the input model as pre-processing, and ignores annotations arguments during precondition checking (issue #736).
- The CIF controller properties checker no longer supports models without automata, which was already reported through warnings before (issue #621).
- The CIF controller properties checker no longer supports models with automata and invariants that are kindless or requirements. It only supports automata with invariants that are plants or supervisors (issue #621).
- The CIF controller properties checker now adds the results of the performed checks to a `controller:properties` annotation, adapting the input model and writing it to a file. By default the output file is named after the input file, with its `.cif` file extension replaced by `.checked.cif`. Use the tool's new *Output file* option to change the output file path (issues #621, #823 and #853).
- The CIF controller properties checker now uses consistent names for the tool itself in the command line help, option dialog, error messages and documentation (issue #621).
- The CIF controller properties checker now has slightly improved console output about the results of the performed checks (issue #621).
- The CIF controller properties checker now stops earlier if no checks are enabled (issue #621).
- The CIF controller properties checker now has improved warnings for specifications without events, as it now also checks for no uncontrollable events, and it now bases the checks on the alphabet of the specification, rather than event declarations (issue #621).
- The CIF controller properties checker now has improved precondition checking, as well as improved output in case of precondition violations (issues #424, #621 and #736).
- The CIF controller properties checker now has improved performance for its non-determinism check that is performed on the input model for the confluence and finite response checks (issue #424).
- The CIF controller properties checker documentation has been split into multiple pages and has various improvements (issues #424, #621 and #736).
- The CIF to mCRL2 transformation now removes all annotations from the input model as pre-processing (issue #736).
- The CIF to Supremica transformation now has improved precondition violation markers for reporting unsupported partial variable assignments (issue #790).
- The CIF to Supremica transformation now removes all annotations from the input model as pre-processing (issue #736).

- The CIF to UPPAAL transformation now have improved precondition violation markers for reporting unsupported partial variable assignments (issue #790).
- The CIF to UPPAAL transformation now removes all annotations from the input model as pre-processing (issue #736).
- The CIF to yEd transformation now removes all annotations from the input model as pre-processing (issue #736).
- The CIF event disabler now checks the output model for annotation constraint violations (issue #823).
- The CIF button/lamp/timer example's ToolDef script for code generation has been slightly improved (issue #736).
- The CIF benchmarks overview generation script now uses row-based averages also for relative values (issue #887).
- The CIF benchmarks overview generation script now has improved weighted averages, preventing skewing of the results by a few models that are very difficult to synthesize (issue #887).
- The CIF documentation on SBE has various improved descriptions related to verification (issue #828).
- The CIF documentation has various small improvements and fixes (issues #410, #424, #593, #621, #628, #691, #736, #804, #810, #811, #830 and #852).

6.11. Version 3.0 (2024-03-31)

Language changes:

- SVG input mappings can now either assign values to input variables (new feature), or map to an event to take (existing feature). This new feature is currently an experimental work-in-progress language feature, and is not yet supported by all CIF tools (issues #227 and #787).
- The syntax for addressables (the left hand side of an assignment) has changed to allow names like `automaton.variable`, but this is only meant to be used for SVG input mappings. It should be avoided for assignments in functions or on edges (issue #227).
- Annotations may now be added to constants, as well as discrete, continuous and algebraic variables. Annotations are still an experimental work-in-progress language feature. Their design may change in a backward incompatible manner (issue #593).

New features:

- All CIF tools now support reading `.cifx` files, CIF files in an XML format. CIF tools that write CIF specifications also support writing `.cifx` files. See the CIF reference manual for more information (issues #701 and #731).
- The CIF to CIF transformer now features a new *Remove annotations* transformation (issue #735).
- The CIF benchmark models have been extended with the `mri_event`, `mri_state` and `wafer_scanner` benchmark models (issue #364).

Improvements and fixes:

- All tools that produce empty CIF files now produce CIF files with an explicit comment indicating that the CIF specification is empty (issue #768).
- The CIF simulator no longer crashes on print output to a file in a non-existing directory, but instead now creates the missing directories (issue #730).
- The CIF data-based synthesis tool has internally been changed significantly, to in the future allow reusing the BDD-related parts of it in other tools. As a result of this, the tool has some changes in error messages for invalid option values (issue #706).
- The CIF data-based synthesis tool's *Supervisor name* option no longer defaults to `sup` if explicitly given an empty option value. The default of the option is still `sup`, but setting the option value to empty now gives an error (issue #269).
- The CIF data-based synthesis tool's warnings for plants that reference requirement state had typos that have been fixed (issue #750).
- The CIF code generator now has more readable output in case of precondition violations, as it has been migrated to the new precondition checking framework introduced earlier already for some of the other CIF tools (issue #424).
- The CIF code generator now supports `doc` annotations on constants, as well as discrete, continuous and algebraic variables. The comments in the generated C, Java, JavaScript, HTML, and Simulink code for `doc`-annotated constants, as well as discrete and algebraic variables now include the documentation from their documentation annotations. The comments in the generated C, Java, JavaScript, and HTML code for `doc`-annotated continuous variables now include the documentation from their documentation annotations (issue #593).
- The CIF code generator now has improved comments for algebraic variables in generated C and Simulink code (issue #593).
- The CIF code generator now generates correct JavaScript and HTML code for tuples (issue #715).
- The CIF code generator now generates correct JavaScript and HTML code for references to parameters and local variables of internal user-defined functions (issue #715).
- The CIF code generator now generates correct JavaScript and HTML code for the `acos`, `asin`, `atan`, `cos`, `exp`, `ln`, `log`, `sin`, and `tan` standard library functions (issue #715).
- The CIF code generator now generates correct JavaScript and HTML code for the string representations of enumeration literals, for instance in error messages, output of `fmt`, output of print declarations, and state logging (issue #715).
- The CIF code generator now generates correct JavaScript and HTML code for string escaping, for instance for log output, certain runtime errors, and state printing (issue #715).
- The CIF code generator now generates correct JavaScript and HTML code for constants whose value is defined using other constants (issue #715).
- The CIF code generator now generates JavaScript and HTML code for SVG copy and move declarations (issue #715).
- The CIF code generator now generates JavaScript and HTML code that can log the initial state and states reached after event transitions. This additional logging is disabled by default (issue #715).
- The CIF code generator now generates JavaScript and HTML code that include variables to

enable or disable state and transition logging (issue #715).

- The CIF code generator now generates JavaScript and HTML code that takes into account the target of print declarations (issue #715).
- The CIF code generator now generates HTML pages with a 'Settings' menu, in which state logging (new feature) and transition logging (existing feature) can be enabled or disabled (issue #715).
- The CIF code generator now generates HTML pages with a log panel that has improved performance (issue #715).
- The CIF code generator now generates HTML pages with a log panel that has a horizontal scrollbar, rather than wrapping long lines (issue #715).
- The CIF code generator now generates HTML pages with less horizontal margin (issue #715).
- The CIF code generator now generates HTML pages that apply proper HTML escaping for the log panel (issue #715).
- The CIF code generator now generates HTML pages that consistently use Unix newlines for the log panel (issue #715).
- The CIF code generator now generates HTML pages that use a queue for SVG input clicks, preserving the order of the clicks, properly deciding the event to take on the current state, ignoring clicks for non-enabled events (printing a warning about it), and handling each click exactly once (issue #715).
- The CIF code generator now generates HTML pages where hovering over interactive SVG elements works correctly in case there are multiple SVG images that have interactive elements with the same ids (issue #715).
- The CIF code generator now generates improved and simpler JavaScript and HTML code for internal user-defined functions (issue #715).
- The CIF code generator now generates comments and reports with the original CIF names for location pointer variables (issue #715).
- The CIF code generator no longer wrongly reports certain warnings when generating Simulink code, for warnings on local variables and arguments of internal user-defined functions that should only be reported for discrete variables. The warning messages have also been improved (issue #424).
- The CIF code generator documentation has some fixes, improvements and extensions (issues #424 and #715).
- The (still experimental) new CIF PLC code generator now has an option to let the generator decide the best way to handle enumerations, based on the configured target PLC model (issue #679).
- The (still experimental) new CIF PLC code generator has various fixes related to code generated for variables, timers and functions (issue #679).
- The (still experimental) new CIF PLC code generator has various improvements and fixes related to names in the generated code, including among others better handling of keywords, not generating names with multiple consecutive underscores, and improved naming of edge selection variables (issue #679).

- The (still experimental) new CIF PLC code generator now generates improved and extended comments in the generated PLC code, for the part of the code related to executing transitions (issue #782).
- The (still experimental) new CIF PLC code generator has improved precondition checking for continuous variables, with better messages in case of unsupported specifications (issue #681).
- The CIF to CIF transformation to eliminate 'if' updates now not only eliminates 'if' updates on edges, but also for SVG input mappings with updates (issue #227).
- The CIF to CIF transformation to simplify values now not only simplifies the guards of 'if' and 'elif' updates on edges, but also for SVG input mappings with updates (issue #227).
- The CIF button/lamp/timer example has been fixed such that code can once more be generated (issue #772).
- The CIF button/lamp/timer and FIFO examples have formatting improvements for their ToolDef scripts (issue #773).
- The CIF ball sorting benchmark model source information had a typo that has been fixed (issue #751).
- The CIF production cell benchmark model has a fixed comment (issue #767).
- The CIF language tutorial 'Variable overview' lesson now also includes input variables (issue #792).
- The CIF documentation breadcrumbs are now consistent with the table of contents (issue #38).
- The CIF documentation table of contents now has proper links for items that link to sections on pages rather than to entire pages (issue #38).
- The CIF website and documentation have various small improvements and extensions (issues #38, #227, #715, #727, #752, #756, #764 and #766).

6.12. Version 2.0 (2023-12-22)

Language changes:

- Annotations may now be added to locations of automata. Annotations are still an experimental work-in-progress language feature. Their design may change in a backward incompatible manner (issue #593).
- A single element of a CIF specification may now be annotated multiple times with the same annotation (typically with different arguments). Annotations are still an experimental work-in-progress language feature. Their design may change in a backward incompatible manner (issue #692).
- The CIF toolset now comes bundled with the `state` annotation. It is currently an experimental work-in-progress annotation. Its design may change in a backward incompatible manner (issues #687 and #697).
- The CIF type checker now warns about certain duplicate state invariants (issue #542).

New features:

- A new CIF PLC code generator has been added to the CIF toolset. The new PLC code generator is

currently being developed, and should be considered experimental. In due time, it will replace the current stable CIF PLC code generator (issues #397, #676 and #679).

- The CIF code generator can now generate HTML files. The HTML files allow executing the model in a browser. The HTML code generation is currently an experimental feature (issues #272 and #715).
- The CIF code generator can now generate JavaScript code. The JavaScript code generation is currently an experimental feature (issues #272 and #715).
- The CIF explorer now generates an output CIF model with state annotations. This can be disabled using the tool's new *Add state annotations* option (issues #687 and #698).
- The tools from the event-based toolset that output CIF models, now generate CIF models with state annotations. This can be disabled using the tools' new *Add state annotations* options (issue #708).
- The CIF documentation now includes a history page, that explains the history of CIF, as well as what 'CIF' used to stand for (issue #669).

Improvements and fixes:

- All CIF tools that produce an output CIF file, no longer produce an invalid CIF file when a model contains integer types that only support the minimum integer value (issue #689).
- The CIF type checker now produces improved error messages in case of a mismatch between an argument of a component instantiation and the corresponding parameter of the instantiated component definition (issue #672).
- The CIF simulator no longer crashes when simulating models with large expressions (issue #611).
- The CIF simulator now truncates messages for runtime errors that include large expressions (issue #661).
- The CIF to CIF transformation to 'add default initial values' now produces correct results for models with discrete variables with an integer type that only supports negative integer values (issue #689).
- The CIF to CIF transformation to 'simplify values' now produces correct results for models with integer expressions that are simplified to the minimum integer value (issue #689).
- The CIF merger now produces correct models when merging models with annotations (issue #680).
- The CIF explorer no longer crashes for models with discrete variables that have a function type and that are not explicitly initialized (issue #690).
- The CIF explorer no longer crashes for models with discrete variables with integer types that only support the minimum integer value (issue #689).
- The CIF explorer no longer crashes for models with expressions that evaluate to the minimum integer value (issue #689).
- The CIF explorer documentation has some fixes, improvements and extensions (issue #687).
- The CIF controller checker now produces more debug output (if debug output is enabled), to be able to better track its progress (issue #686).

- The CIF controller checker is now more responsive to termination requests (issue #686).
- The CIF to mCRL2 transformer now supports CIF models with logical implication binary operators (issue #667).
- The CIF code generator now generates better-formatted Java code for checking range violations (issue #272).
- The CIF code generator now generates valid Java and Simulink code for models without enumerations and without any automata that have at least two locations (issues #272 and #715).
- The CIF code generator no longer crashes for certain models with more complex initialization. Previously, if an automaton had multiple locations with initialization predicates that contained variables, but only one of those locations was an initial location, the code generator could crash (issue #683).
- The CIF PLC code generator (stable version) no longer crashes for certain models with more complex initialization, if its *Simplify values* option is disabled. Previously, if an automaton had multiple locations with initialization predicates that contained variables, but only one of those locations was an initial location, the code generator could crash (issue #684).
- The 'lithography_init' benchmark model has been changed, removing a duplicate invariant (issue #712).
- The CIF language tutorial has various improvements and extensions for the documentation about annotations (issue #593).
- The CIF language reference manual now includes additional detailed information about annotations (issue #593).
- The CIF tool manual now indicates for each tool which annotations it supports, and how it handles them (issue #593).
- The CIF documentation has various small improvements and fixes (issues #593, #648, #669, #682, #687 and #708).

6.13. Version 1.0 (2023-09-30)

Language changes:

- The CIF language now features annotations, that can be used to annotate elements of the specification with extra information. Annotations are currently an experimental work-in-progress language feature. Their design may change in a backward incompatible manner (issue #593).
- The CIF toolset now comes bundled with the `doc` annotation. It is currently an experimental work-in-progress annotation. Its design may change in a backward incompatible manner (issue #593).

Improvements and fixes:

- The CIF simulator no longer deadlocks if an element of an SVG image is clicked, and the corresponding CIF/SVG input mapping maps to an event that is disabled in the current state. Instead, it now prints a warning and ignores the click (issue #469).

- The CIF controller checker had a performance regression since version 0.7, in case finite response is checked, and confluence is not checked. This performance regression has been fixed (issue #639).
- The CIF controller checker now requires at least one check to be enabled (issue #639).
- The CIF data-based synthesis tool's workset algorithm now has improved edge selection heuristics, improving the performance of the workset algorithm (issues #520 and #605).
- The CIF data-based synthesis tool's workset algorithm is no longer considered experimental (issue #655).
- The CIF data-based synthesis tool's workset algorithm documentation has been improved, to better explain when to use and not to use the workset algorithm (issue #655).
- The CIF to Supremica transformation now correctly transforms multiple guards on an edge. Multiple guards are now combined into a single conjunction (issue #620).
- The CIF merger does not allow merging two input variables with the same annotations (issue #593).
- The CIF code generator now supports `doc` annotations on input variables. The comments in the generated C, Java and Simulink code for `doc`-annotated input variables, now include the documentation from their documentation annotations (issue #593).
- The CIF event disabler tool now properly closes the event names file after reading it, preventing a resource leak (issue #649).
- The CIF explorer command line script no longer crashes due to a failure to find the application to execute (issue #658).
- The 'festo' benchmark model has been changed, removing a duplicate invariant (issue #646).
- The FIFO CIF example now uses SVG input, rather than GUI input. It is now possible to click the arrows in the SVG image to control the simulation (issue #639).
- The Wolf/Goat/Cabbage CIF example now has simpler CIF/SVG input mappings (issue #639).
- The CIF documentation pages for the CIF code generator now use syntax highlighting for C and Java code (issue #569).

6.14. Version 0.10 (2023-06-30)

Language changes:

- Switch expressions with only a single `case` or `else` are now deprecated (issue #612).
- The CIF type checker now allows omitting the `else` part of `switch` expressions in more situations, when it can be statically determined that the `switch` expression is complete without it. The type checker has also been improved, regarding detection of duplicate and missing switch cases, and superfluous `else`s (issue #548).

Deprecations and removals:

- The curly-brackets syntax for declaring enumerations was deprecated in 2014 and has now been removed. Use the syntax without curly brackets instead (issue #613).

- The CIF PLC code generator's *Eliminate enumerations* option was deprecated in release v0.2 and has now been removed. Use the *Convert enumerations* option instead, which allows for more control (issue #613).
- The CIF to CIF transformation to eliminate enumerations (`elim-enums`) was deprecated in release v0.2 and has now been removed. Use the transformation to convert enumerations to integers (`enums-to-ints`) instead, as it has the same functionality (issue #613).

New features:

- The CIF data-based synthesis tool has a new *Edge granularity* option to configure the granularity of edges to use during synthesis. Previously, the linearized edges were used 'as is'. Now, they are merged per event, to obtain a single edge per event. The option can be used to restore the old behavior. The new edge granularity has been shown to reduce both the memory usage and time required to perform synthesis, but this may depend on the model being synthesized. For more information, see the documentation of the new option (issue #586).
- The CIF data-based synthesis tool has a new *Fixed-point computations order* option to configure the order in which the fixed-point computations are to be performed during synthesis. By default, the fixed-point computations have the same order as before. First the non-blocking states are computed, then the controllable states, and finally the reachable states (the reachable states are still only computed if the *Forward reachability* option is enabled). Using the new option, the order can be changed. This may significantly improve synthesis performance, but the effect greatly depends on the model being synthesized (issue #609).
- The CIF data-based synthesis tool now features an experimental workset algorithm to dynamically determine the order in which edges are considered during reachability fixed-point computations. It is currently considered to be experimental and is disabled by default. It can be enabled using the new *Edge workset algorithm* option. It can only be used when a per-event edge granularity is configured, and duplicate events are allowed in the edge order (issues #520 and #602).

Improvements and fixes:

- The position information of invariants has been improved. All CIF tools that may report on invariants give improved file position information, for example in type checker errors and precondition violation reports (issue #591).
- The CIF simulator's command line script has been fixed for handling dark theme detection, preventing crashes and improving dark theme support, when using the simulator's plot visualizer or interactive GUI input component (issue #567).
- The CIF simulator's SVG visualizer *Save as* dialog now properly starts in the directory that contains the SVG file, also on Windows. And it now properly handles paths with spaces and other special characters in them (issue #221).
- The CIF explorer now warns about exploring specifications with requirements (issue #585).
- The CIF event-based synthesis toolset's command line scripts now have correct information to start the tools, where previously they always crashed (issue #567).
- The CIF event-based synthesis analysis tool's command line script has been fixed for handling dark theme detection, preventing crashes and improving dark theme support (issue #567).

- The CIF event-based synthesis analysis tool's command line script has been fixed to not immediately terminate after starting the tool (issue #567).
- The CIF event-based synthesis tool now warns about synthesizing specifications without requirement automata (issue #553).
- The CIF data-based synthesis tool no longer prints the number of states in the controlled system as part of its debug output. Instead, this information is now one of the statistics of the *Statistics* options. The new statistic is disabled by default, and if enabled the statistic is now printed to normal output. The output itself has some whitespace changes (issue #526).
- The CIF data-based synthesis tool now has better performance for converting updates of the CIF specification to their internal BDD representations (issue #580).
- The CIF data-based synthesis tool now has slightly better performance due to skipping some unnecessary calculations for applying edges during reachability fixed-point computations (issue #520).
- The CIF data-based synthesis tool now enforces state plant invariants like it does state requirement invariants, when they are applied per edge. This may improve synthesis performance for models with state plant invariants (issue #523).
- The CIF data-based synthesis tool's *BDD library operation cache ratio* option has been fixed. It now functions as documented, rather than having the inverse effect. It now also has improved precision (issue #610).
- The CIF data-based synthesis tool's legacy hyper-edge creation algorithm now properly considers invariants in locations for variable ordering. Previously, for every location, the invariants of the automaton that contains the location were considered, rather than considering the invariants of the location itself. This change could affect synthesis performance (issue #377).
- The CIF data-based synthesis benchmarking scripts no longer produce debug output when performing synthesis, making benchmarking faster and producing less output, while still reporting the platform-independent metrics and the number of states in the controlled system (issues #526 and #564).
- The CIF data-based synthesis documentation has several improvements and extensions (issues #609 and #610).
- The CIF controller property checker's finite response check may now have improved performance (issue #583).
- The CIF to CIF transformation to merge all enumerations of the specification into a single enumeration has been changed. It now converts the last case of all `switch` expressions to an `else`, for switch expressions that switch over a value that has a type that includes an enumeration type. Hereby, the keys of these last cases are removed from the model (issue #548).
- The CIF language tutorial has an additional lesson on `if` and `switch` expressions (issue #598).
- The CIF language tutorial lesson on function statements has been improved and extended (issue #598).
- The website has been fixed to ensure all version-specific URLs are now properly replaced by non-version-specific ones when a version-specific website is promoted to become the main website. Previously, a few URLs remained version-specific (issue #577).

6.15. Version 0.9 (2023-03-31)

New features:

- The CIF type checker now warns about certain duplicate state/event exclusion invariants (issue #299).
- The CIF transformer has a new additional CIF to CIF transformation, to remove unused events (issue #525).
- The CIF data-based synthesis tool variable ordering configuration has been generalized and extended. This includes the addition of a new *BDD advanced variable ordering* option. It offers much more flexibility in configuring variable ordering, including configuration of the order in which to apply various algorithms, and configuration of the settings to use per algorithm. As a result of the changes, the debug output has been changed considerably. See the documentation of the new option for more information (issues #378 and #530).
- The CIF data-based synthesis tool *Edge order* option has been replaced by two separate *Edge order for backward reachability* and *Edge order for forward reachability* options. This allows to specify the orders separately for forward and backward reachability computations, which may improve synthesis performance. The defaults have not been changed (issue #492).
- The CIF data-based synthesis tool has a new *Edge order duplicate events* option to allow duplicate events in custom event orders (issue #426).
- The CIF data-based synthesis tool now has a *State requirement invariant enforcement* option, adding an alternative second approach to apply state requirement invariants during synthesis. Both approaches have potential benefits and drawbacks, making for a trade-off between their various effects. Which approach is most efficient depends on the model. The default has not been changed. As a result of the changes, the debug output of the tool has changed as well (issue #198).
- The CIF data-based synthesis tool statistics option has been extended to allow measuring the maximum amount of memory used during synthesis. See the documentation for more information on how to best use this new feature (issue #531).
- The CIF data-based synthesis tool benchmarking scripts have newly added functionality. The settings to configure the configurations to benchmark have been generalized, making it more powerful. It is also possible to execute benchmarking for only a subset of the configurations. The generated HTML overviews now allow switching between absolute and relative values, feature extra columns with average values, and some of the columns can now be sorted (issues #538, #539 and #551).
- The CIF benchmarking set of models now includes more than 10 additional benchmarking models (issues #364, #522, #532 and #537).
- The CIF bridge example model now includes a readme file that describes several simulation scenarios (issue #489).

Improvements and fixes:

- The CIF type checker now reports various errors at better positions, including for function calls to standard library functions (issue #454).

- The CIF simulator state visualizer performance has been greatly improved (issue #485).
- The CIF simulator error message for unsupported specifications with time-dependent state invariants has been improved (issue #513).
- The CIF data-based synthesis tool now detects reachability fixed points sooner, reducing fixed point computations by at most one iteration. The effect on synthesis performance greatly depends on the model (issue #493).
- The CIF data-based synthesis tool no longer skips variable ordering if representations of relations between variables are empty, but unused (issue #378).
- The CIF data-based synthesis tool options that influence the variable ordering have some new defaults. The DCSH variable ordering algorithm is no longer considered experimental, and is now enabled by default. The *BDD hyper-edge creation algorithm* option has a new `default` value that is set by default. It uses the linearized hyper-edges for the FORCE and sliding window algorithms, while for all other variable orderers the legacy hyper-edges are still used. These changes to the default variable ordering configuration have been shown to improve the out-of-the-box performance of data-based synthesis in many cases, especially for models that take longer to synthesize or require more memory to synthesize. However, the effect greatly depends on the model being synthesized, and for some models synthesis using default settings may now be slower (issues #378 and #379).
- The CIF data-based synthesis tool documentation has various small improvements (issues #426, #492 and #531).
- The CIF data-based synthesis tool documentation related to variable ordering has been split into multiple pages, and has been extended and improved (issues #378 and #530).
- The CIF data-based synthesis tool error messages for invalid custom event orders have been improved (issues #426 and #492).
- The CIF data-based synthesis tool *BDD variable order* option has been renamed to *BDD initial variable ordering*. The command line name of the option has not been changed (issue #530).
- The CIF data-based synthesis tool *BDD initial variable ordering* option now supports values with leading and trailing whitespace (issue #378).
- The CIF data-based synthesis tool termination request detection has been improved, by reducing the number of times it unnecessarily checks for termination requests (issue #198).
- The CIF data-based synthesis tool benchmarking scripts have been improved in various ways. The scripts and generated overviews now use improved and consistent terminology. The script to execute benchmarks no longer warns about events in the models that are disabled. The script to generate HTML overviews is more robust in handling unknown values, partial or missing information, as well as inconsistent and duplicate information. The generated HTML overviews feature more colors to more easily see which configurations work better or worse, have various styling improvements, contain additional information, and warn about potential issues in the experiment data. There are some other smaller improvements as well (issues #522, #538, #539 and #551).
- The CIF event-based synthesis tool now allows synthesis for specifications without requirements (issue #544).
- The CIF event-based language equivalence check tool now prints a counter example path if the two automata are not language equivalent (issue #413).

- The CIF code generator has small improvements to the Java code that it generates (issue #450).
- The CIF PLC code generator option description for the *PLC number bits* option has been fixed (issue #418).
- Various CIF tools have completely redesigned output in case of precondition violations. The new output groups similar violations, has improved readability, and provides more context and information about where in the model the violations occur. There are some other smaller improvements as well. The new output is produced by the CIF code generator, CIF PLC code generator, CIF to UPPAAL transformer, and the CIF to Supremica transformer (issue #454).
- The CIF tutorial has various small fixes and other improvements (issues #501, #502, #503, #504, #506, #509, #510 and #512).
- The CIF documentation has some improvements related to 'file separator' terminology (issue #450).

6.16. Version 0.8 (2022-12-21)

Language changes:

- Improved type checking for the `=` and `!=` binary expressions. The type checker now not only uses the type of the left operand as type hint for the right operand, but also the type of the right operand as type hint for the left operand, in case the former is not sufficient. This allows for instance type widening for the left operand to be derived from the right operand, such that both expressions `x = 1` and `1 = x` are now supported, with `x` a `real`-valued variable (issue #446).

New features:

- The CIF data-based synthesis tool now has a second hyper-edge creator, the linearized hyper-edge creator, which may improve performance. The legacy hyper-edge creator is still used by default. The hyper-edge creator can be configured using the new *BDD hyper-edge creation algorithm* option (issue #380).
- A new CIF example model has been added (issue #465).
- A new CIF benchmarking model has been added (issue #451).
- The CIF benchmarking models now come with scripts to easily benchmark data-based synthesis (issues #375 and #455).

Improvements and fixes:

- Normally, `.cif` files are opened with the CIF text editor. Large files are now however opened with the default non-CIF text editor to avoid performance issues. You can open a file in an editor of your choosing by right clicking it and selecting **Open With** and then selecting the editor of your choosing, or choosing **Other...** to open a dialog to choose from a larger selection of editors (issue #199).
- The CIF simulator no longer creates a directory named `cifcode` in the root of your drives or file systems when using the Eclipse Compiler for Java (ECJ) to compile Java code (issue #176).
- The CIF simulator state visualizer performance has been slightly improved (issue #477).
- The CIF simulator state visualizer's *Value* column width adjustments are now less 'jittery' (issue

#477).

- The CIF simulator plot visualizer and GUI input component now have theming support, and come with a dark theme in addition to the existing light theme. They now automatically use their dark theme when the Eclipse built-in dark theme is used, and use a light theme otherwise (issue #417).
- The CIF simulator plot visualizer now has slightly darker gridlines in the light theme, for enhanced contrast (issue #417).
- The CIF event-based synthesis analysis tool now has theming support, and comes with a dark theme in addition to the existing light theme. It now automatically uses its dark theme when the Eclipse built-in dark theme is used, and uses a light theme otherwise (issue #417).
- The CIF event-based synthesis analysis tool now has slightly wider colored boxes that distinguish between disabled and different types of enabled choices, for enhanced visual clarity (issue #417).
- The CIF to Supremica transformation precondition check has improved output. The preconditions themselves have not changed (issues #368, #424, #442 and #470).
- The CIF to UPPAAL transformation precondition check has been migrated to the new precondition check framework, which is already used by the CIF to Supremica transformation, resulting in improved output. The preconditions themselves have not changed (issue #424).
- The CIF PLC code generator precondition check has been migrated to the new precondition check framework, which is already used by the CIF to Supremica transformation, resulting in improved output. The preconditions themselves have not changed (issue #457).
- Very long lines in the *Console* view, for instance outputted by the CIF simulator, and long lines in text editors, for instance long lines in synthesized CIF models, now render correctly on Windows (issue #76).
- The CIF documentation has been updated to account for changes in recent Inkscape versions (issue #357).
- The CIF data-based synthesis documentation has been split into multiple pages, as it was getting quite long (issue #460).
- The CIF data-based synthesis documentation on supported updates, expressions and predicates has been improved (issue #459).
- Some very small other documentation improvements (issues #199, #438 and #460).

6.17. Version 0.7 (2022-09-30)

New features:

- The CIF controller property checker can now also check whether controllers satisfy the confluence property. As the checker now has multiple properties that it can check, each property check can be individually enabled and disabled, and all checks are enabled by default (issue #145).
- The CIF examples contain a new bridge example, to showcase the real-world usage of CIF for synthesis-based engineering (issue #419).

- Two new CIF benchmarking models have been added (issue #364).
- The CIF text editor now has theming support, and comes with a dark theme in addition to the existing light theme. The text editor now automatically uses its dark theme when the Eclipse built-in dark theme is used, and uses a light theme otherwise. The text editor theming behavior can be configured via the Eclipse Preferences dialog (issue #347).

Improvements and fixes:

- The CIF event-based synthesis tool now records removal of non-reachable locations in the synthesis dump file. This new information in synthesis dump files is now also taken into account by the CIF event-based synthesis analysis tool (issue #404).
- The CIF event-based language equivalence check tool now produces correct counter examples (issues #297 and #405).
- The CIF controller checker output has been improved in terms of readability (issue #145).
- The CIF controller checker option descriptions and UI have been slightly improved (issue #145).
- The CIF controller checker documentation has been improved (issue #145).
- The CIF to Supremica transformation precondition check has improved output and no longer crashes on reporting certain precondition violations. The preconditions themselves have not changed (issues #398 and #416).
- The CIF text editor light theme's default color has changed from a near-black slightly-brown color to pure black (issue #347).
- The CIF to yEd transformer syntax highlighter's default color has changed from a near-black slightly-brown color to pure black (issue #347).

6.18. Version 0.6 (2022-07-07)

Language changes:

- Invariants can now be given a name (issue #323).

New features:

- Added a CIF benchmark models import wizard, with several benchmarking models, similar to the CIF examples import wizard (issue #364).
- The CIF data-based synthesis tool now warns about input models where plants refer to requirement state. The new functionality is enabled by default, but can be disabled using an option (issue #311).
- The CIF data-based synthesis tool features the DCSH algorithm as new additional variable ordering algorithm. It is considered experimental for now, and therefore disabled by default, but can be enabled using an option (issue #196).

Improvements and fixes:

- The CIF data-based synthesis tool's automatic variable ordering has various internal improvements in preparation for further upcoming improvements. As a result the debug output

has been extended and changed a bit (issues #196 and #371).

- The CIF data-based synthesis tool's automatic variable ordering documentation has been improved and extended (issue #196).
- The CIF data-based synthesis tool has some small consistency improvements for its command line option descriptions (issue #311).
- Some improvements to the documentation of the CIF to CIF linearize merge/product transformations (issue #354).
- The CIF to Supremica transformation precondition check has been redesigned, producing different output. The preconditions themselves have not changed (issue #370).
- Links in the documentation to non-CIF ESCET documentation webpages now use version-specific URLs (issue #386).
- Improved bibliography references in documentation (issue #365).
- The issue numbers in the release notes now link to the corresponding GitLab issue (issue #396).
- Small website style improvements (issue #367).

6.19. Version 0.5 (2022-03-29)

New features:

- The CIF documentation contains an extensive new section on synthesis-based engineering of supervisory controllers (issues #325 and #334).
- The CIF examples contain a new FIFO example, to showcase the power of supervisory controller synthesis. It is also described in the new synthesis-based engineering documentation (issue #325).
- The CIF PLC code generator now supports S7 output for SIMATIC controllers (issue #142).
- The CIF data-based synthesis tool now supports state plant invariants (issue #107).
- The CIF to CIF transformer has several new transformations, to anonymize the names of all named objects, convert controllable events to uncontrollable ones and vice versa, and remove unused algebraic variables and their equations (issues #306, #308 and #320).

Improvements and fixes:

- The CIF type checker no longer produces the wrong warning for globally disabled event that are monitored, for events referred to via component parameters or component instantiations (issue #300).
- The CIF type checker no longer produces false positive/negative warnings and errors related to the use of events in component definitions/instantiations. This applies to the following checks: duplicate events in alphabets, duplicate monitor events, duplicate events on edges, events on edges not in alphabets, monitoring events not in alphabets, events in alphabet not on any edge, monitored event not on any edge, and monitoring an empty alphabet. It also has several other small improvements related to these checks, including improved warning messages and reporting the warnings at better positions. As a result of the changes, problems are no longer reported for non-instantiated component definitions (issues #298 and #301).

- The CIF simulator interactive GUI input component now has a horizontal scrollbar from the start, if applicable (issue #217).
- The CIF simulator being terminated by the red stop button no longer leaves the interactive GUI input component buttons enabled (issue #274).
- The CIF PLC code generator PLC number bits option now has a new automatic mode. The default remains 64-bit for PLCopen XML, IEC 61131-3 and TwinCAT output (issue #142).
- The CIF PLC code generator now generates correct assignments to temporary variables for multi-assignments (issue #282).
- The CIF PLC code generator now generates calls to the `NOT` function without formal argument names (issue #286).
- The CIF documentation has several small improvements (issues #271, #324 and #326).
- The release notes for each version now contain the release date, with the exception of milestone releases and release candidates (issue #314).

6.20. Version 0.4 (2021-12-17)

Language changes:

- Component parameters can now be used as values, for instance as arguments to component instantiations. See the CIF language tutorial lesson on group definitions for more information (issue #222).

New features:

- The CIF simulator now supports simulating specifications with input variables. Use the *CIF specification initialization* option to specify the initial value of input variables. Support for modifying the value of input variables during simulation is planned for future versions (issue #203).
- The CIF data-based synthesis tool now has an *Edge order* option to influence the order in which edges are considered, which can significantly influence the performance (issue #197).

Improvements and fixes:

- Introduced a brand new website (issue #35).
- Many website URLs have changed due to various website structure changes (issues #35 and #73).
- Various documentation/website textual improvements, style improvements and other changes (issues #35, #54, #188 and #236).
- Various CIF tools now correctly handle a tuple projection index that is a constant reference with a name that is hidden by a tuple field name. In particular the CIF pretty printer now generates correct references for such indices (issue #258).
- The CIF to CIF transformation to eliminate component definition/instantiation now correctly handles cases where component definitions are referred to via component instantiations (issue #244).

- The CIF type checker now warns about some overly-convoluted references (issue #234).
- The CIF simulator interactive GUI input mode now is fully responsive to termination requests when asking users for the next transition to take (issue #216).
- The CIF simulator SVG visualizer functionality to save the image as an SVG file no longer crashes (issue #205).
- The CIF simulator plot visualizer and SVG visualizer functionality to save as an image no longer ask duplicate overwrite questions (issue #223).
- The CIF simulator no longer crashes when showing an SVG visualizer if the SVG file can't be loaded (issue #207).
- The CIF simulator trajectory data file option description has been improved (issue #248).
- The CIF data-based synthesis tool now supports collecting additional statistics, i.e., maximum used BDD nodes, continuous BDD performance, and BDD cache statistics (issue #171).
- The CIF data-based synthesis tool statistics option now has different option value names for various statistics. This is a backward incompatible change. Use `bdd-gc-collect` instead of `bdd-gc`, and `bdd-gc-resize` instead of `bdd-resize` (issue #235).
- The CIF data-based synthesis tool documentation has various small improvements (issue #171).
- The CIF to CIF linearization transformation now preserves more of the model structure. Groups are no longer eliminated, I/O file declarations are no longer pushed inwards, I/O declarations are no longer merged, and enumerations are no longer merged. The original automata are replaced by groups to allow keeping most declarations (e.g., algebraic variables) declared within them in their original scope, allowing them to keep their original absolute identities. Initialization predicates, marker predicates and invariants are also kept (close to) where they were. Linearization may now result in models with scope absolute references. Some warning messages now include proper absolute names. The documentation has been improved as well (issues #155 and #245).
- The CIF to CIF linearization transformation now uses temporary location pointer variable names that are less likely to lead to name clashes and renamings (issue #245).
- The CIF to CIF transformation to eliminate the use of locations in expressions now generates location pointer variables and enumerations with simpler names. This also affects the CIF to Supremica transformation (issue #245).
- The CIF to CIF transformation to eliminate the use of locations in expressions has improved renaming warnings (issue #245).
- The CIF PLC code generator has some backward incompatible name changes in the generated PLC code due to changes the CIF to CIF linearization transformation (issue #155).
- The CIF PLC code generator has improved absolute names in error messages, generated comments, etc (issue #155).
- The CIF PLC code generator now generates proper PLC names for CIF objects that have a CIF keyword as name (issue #249).
- The CIF code generator has improved absolute names in generated comments, improved absolute original event names, and improved print declarations order in generated code (issue #155).

- The CIF code generator has improved error and warning messages for Simulink code generation (issue #250).
- The CIF code generator now takes into account all Java 11 keywords, literals and identifiers with special meaning for Java code generation (issue #238).
- The CIF code generator now generates simpler and better performing code for (in)equality binary operators that compare enumeration literals (issue #247).
- All CIF tools that support but ignore SVG input declarations now print a warning for specifications that contain SVG input declarations (issue #218).
- The CIF to mCRL2 tool *Generate value actions* option has been improved to not generate any value actions if an empty option value is provided. This is a backward incompatible change. The documentation of the option has also been improved (issue #225).
- The CIF to mCRL2 tool no longer eliminates enumerations to integers, but instead generates mCRL2 sorts for CIF enumerations. This allows referring to the enumeration literals by name when specifying properties to verify (issue #156).
- The CIF to mCRL2 documentation on supported CIF specifications has been improved (issue #156).
- The CIF to yEd transformation is now part of the miscellaneous tools. In the Eclipse ESCET IDE it can now be found in the **CIF miscellaneous tools** sub-menu (issue #188).
- The CIF to yEd documentation has been updated for changes to recent yEd versions (issue #209).

6.21. Version 0.3 (2021-10-01)

Language changes:

- Removed inheritance of a supervisory kind for invariants. This was deprecated over 6 years ago and already led to warnings in the specification. Invariants without a supervisory kind specification (plant, requirement, supervisor) that reside in an automaton or location of an automaton, and where that automaton has a supervisory kind, now no longer inherit the supervisory kind of that automaton. Instead they remain kindless. This is a backward incompatible change (issue #139).

New features:

- CIF controller property checker application added to check supervisory controllers for finite response (issue #122).
- CIF data-based synthesis has a new option to print warnings for events that are never enabled in the input specification or the synthesized controlled system. This new option is enabled by default (issues #108, #150, #162 and #164).
- CIF data-based synthesis now supports boolean constants in predicates (issue #143).
- CIF data-based synthesis now supports switch expressions (issue #148).

Improvements and fixes:

- CIF to CIF transformation to remove requirements no longer supports removing requirement

automata that contain locations with non-requirement (kindless, plant, or supervisor) invariants, preventing the loss of invariants (issue #140).

- CIF to CIF transformation to remove requirements now properly moves invariants from requirement automata to their parents, rather than also removing them (issue #140).
- CIF data-based synthesis simplification of supervisor guards for `false` guards under `false` assumption now correctly results in `true` simplified guards, to indicate the supervisor doesn't enforce a restriction (issue #144).
- CIF data-based synthesis documentation updated to reflect that code generator tools support state/event exclusion invariants (issue #163).
- CIF PLC code generator error message now indicates correct TwinCAT XAE project file path (issue #154).
- CIF type checker improved to report more issues for list projections with impossible bounds, rather than deferring to runtime checks (issue #177).
- The CIF simulator, CIF to UPPAAL transformation, and CIF to mCRL2 transformation are now all under the 'CIF simulation, validation and verification tools' part of right-click menus on CIF files and editors (issue #122).
- The website and Eclipse help now use multi-page HTML rather than a single HTML file, although the website still contains a link to the single-page HTML that allows easily searching the full documentation (issue #36).
- Enabled section anchors for documentation on the website, and disabled section anchors for Eclipse help (issue #36).
- Several small documentation fixes and improvements (issues #36 and #166).

6.22. Version 0.2 (2021-07-07)

Language changes:

- A switch case key must now have a type that is compatible (ignoring ranges) with the type of the switch value, rather than the type needing to be fully contained in the type of the switch value. This is a backward compatible change (issue #105).

Deprecations and removals:

- CIF deprecated enumeration declaration syntax (with curly brackets) now leads to deprecation warnings. The documentation now properly reflects the deprecation as well (issue #45).
- CIF to CIF transformation to eliminate enumerations (`elim-enums`) is now deprecated. Use the transformation to convert enumerations to integers instead, as it has the same functionality (issue #78).
- CIF PLC code generator option to either eliminate enumerations (to integers) or not is now deprecated. Use the new option to choose whether to convert enumerations to constants or integers, or not at all. See the documentation of the PLC code generator for further details (issue #78).
- CIF simulator deprecated 'interactive input mode' (`interactive` value) has been removed. Use

the 'interactive console input mode' instead (issue #81).

New features:

- CIF to CIF transformation to convert enumerations to constants (`enums-to-consts`) and enumerations to integers (`enums-to-ints`) has been added (issue #78).
- CIF explorer now has an option to specify the name of the resulting statespace automaton (issue #55).
- CIF PLC code generator now has an option to choose whether to convert enumerations to constants or integers, or not at all. This replaces the old option to either eliminate enumerations (to integers) or not, which is now deprecated. See the documentation of the PLC code generator for further details (issue #78).

Improvements and fixes:

- Various fixes for handling component definition/instantiation as well as references via component instantiations and component parameters (issues #25, #39 and #66).
- CIF type checker now properly detects cycles for algebraic variables and derivatives of continuous variables defined via equations per location of an automaton (issue #106).
- CIF to CIF 'eliminate the use of locations in expressions' transformation no longer generates location pointer variables for automata with exactly one location (issue #99).
- CIF to CIF linearize transformations no longer generate location pointer variables for automata with exactly one location (issue #99).
- CIF to CIF linearize merge transformation no longer crashes for channels without a sender (issue #56).
- CIF simulator no longer crashes on enumeration literals that have a name that is a Java keyword (issue #104).
- CIF simulator initialization fixed to properly account for algebraic variables and derivatives of continuous variables defined via equations per location of an automaton (issue #106).
- CIF simulator now correctly determines initial value for locations without initialization predicates, when those locations are used in initial values of variables (issue #37).
- CIF simulator no longer crashes on assignments to dictionaries with non-int keys (issue #47).
- CIF simulator no longer crashes when using the Eclipse Compiler for Java (ecj) as Java compiler (issue #46).
- CIF simulator is now more robust for larger numbers of state invariants in components (issue #49).
- CIF data-based synthesis now supports state/event exclusion plant invariants (issue #84).
- CIF data-based synthesis now properly ensures requirement invariants are rebranded as supervisor invariants in the synthesis output. Therefore, simulating a synthesized supervisor no longer gives warnings for simulating requirement invariants (issues #116 and #117).
- CIF data-based synthesis documentation, debug output and warnings improved/fixed regarding invariants vs requirement invariants and controlled vs uncontrolled system (issue #83).

- CIF PLC code generator now supports state/event exclusion invariants (issue #94).
- CIF PLC code generator no longer generates location pointer variables for automata with exactly one location when performing linearization as preprocessing (issue #99).
- CIF PLC code generator now creates output folder if it does not yet exist, rather than giving an error, if IEC 61131-3 output is requested (issue #74).
- CIF PLC code generator documentation now correctly indicates the order in which CIF to CIF transformations are applied (issue #78).
- CIF PLC code generator PLCOpen XML output fixes for PLC tasks (issue #75).
- CIF code generator now supports state/event exclusion invariants (issue #94).
- CIF code generator no longer generates location pointer variables for automata with exactly one location when performing linearization as preprocessing (issue #99).
- CIF to mCRL2 transformation now supports algebraic variables (issue #120).
- CIF to mCRL2 transformation precondition check messages and documentation improvements (issues #59, #60 and #120).
- CIF to Supremica precondition check message wording for discrete variables with multiple potential initial values has been fixed (issue #58).
- CIF to Supremica transformation now supports state/event exclusion invariants (issue #94).
- CIF to UPPAAL precondition check message wording for discrete variables with multiple potential initial values has been fixed (issue #58).
- CIF to UPPAAL transformation now supports state/event exclusion invariants (issue #94).
- CIF to UPPAAL transformation now supports algebraic variables and enumerations (issue #129).
- CIF tutorial updated to remove explanation of list subtraction operator that doesn't exist in the language and implementation (issue #77).
- CIF documentation updated to fix two broken links (issue #80).

6.23. Version 0.1 (2021-04-02)

The first release of CIF as part of the Eclipse ESCET project. This release is based on the initial contribution by the Eindhoven University of Technology (TU/e).

Most notable changes compared to the last TU/e release:

- The names of the CIF command line tools and tools available in ToolDef scripts have changed. For more information, check the list of currently available tools.
- The CIF simulator no longer crashes on code generation.
- The CIF simulator plot visualizer has been re-implemented using different third party libraries.

Chapter 7. Developers

The CIF developers manual is intended only for those who develop or extend the CIF language or tools. It is not intended for end users.

This manual only contains CIF specific information. For general developers information, see the Eclipse ESCET developers manual.

The following information is available for developers:

- How to extract information from a CIF specification
- How to modify the CIF language

7.1. Extracting information from a CIF specification

A CIF specification models the behavior of a system. It contains lots of information, and some of the information may be useful outside the CIF model as well. That raises the question how to extract such information from the model.

A very important property here is reliability. The used approach should always return the information that the specification contains, and not for example silently skip some parts. Also, it should not return information that is not part of the specification.

This is a very hands-on practical guide, with programming examples. For these examples, the Python programming language is used. For further information, links to standard Python libraries are provided, where appropriate. The considerations and approaches described here are however generally applicable to most programming languages. The Python libraries used in this document also exist for most other popular programming languages.

7.1.1. Recognizing text of a CIF specification

A common approach is to see a specification as a sequence of text lines, and use text processing techniques to extract the information. In this approach, the text is searched for relevant key fragments. The needed information is then collected from the found fragments. The technique is quite easy to do, and for text that has a fixed form it works quite well.

Unfortunately, text of a CIF specification does not have a fixed form. For instance, you can break lines at any point, causing a key fragment to get split across several text lines. Also, empty lines can be inserted as well, causing a key fragment to have multiple empty lines in its text. Furthermore, each line can have a comment at the end of each line, causing a key fragment to have pieces of comments in its text.

Comments can also contain the text of a key fragment, but the information in the comment is not an active part of the specification. Similarly, a specification may have string constants with text that looks like a key fragment, but again it is not part of the specification. If the key fragment looks for a keyword, variable or event names in the specification may look like a keyword. For example, `event $event` declares an event named `event`. Writing something this confusing should probably not be done in practice, but it is allowed in CIF and thus a search on key fragments may run into such

cases.

Depending on how the search is done, you may get false positive matches (text is found that shouldn't be matched) and/or false negative (text that should be matched is not found). In all, these complications make the common approach of searching key fragments quite unreliable.

To deal with the complexity of the text of a CIF specification, a *parser* should be used instead. A parser does not look for fragments, but instead reads all text. Also it knows exactly how to handle everything that can be written in a CIF specification. For this reason it cannot be fooled by splitting or inserting lines, or by writing strange comments, text strings or event names.

As a parser needs more knowledge, creating one is more work. If you want to venture in this area, have a look at a list of parser generator tools for Python. For the CIF language however, a parser already exists. It is used by all CIF tools.

Different CIF tools have different needs for information from a CIF specification. Saving only a part of the information of a CIF specification would make the parser useless for tools that need information beyond the saved part. To avoid that, the parser saves *all* information of the CIF specification, as a tree of objects. Each CIF tool selects the particular information from the tree that it needs.

A file with such a tree can be obtained by using the `.cifx` extension for an output file of a CIF tool, instead of the usual `.cif` extension. The CIF reference manual explains it in the CIF XML files section.

The reverse also works, all CIF tools accept files with a `.cifx` extension. In addition, other programs can also read such a file and search in the object-tree for relevant information, without all the complications that exist when searching in CIF specification text.

The next section discusses the object tree in some more detail. Where to find *all* details of the object tree is explained in the CIF meta-model section. In the Getting relevant information from the object-tree section, searching the tree for relevant information is discussed.

7.1.2. Structure of the CIF object-tree

A loaded CIF file is internally represented as a tree of objects. The objects in the tree follow the structure of a CIF specification. The tree starts with a `Specification` object. That object may have `Group` and `Automaton` objects (and also other CIF elements, such as declarations and requirement invariants). Each `Group` object can in turn have more `Group` objects, and so on. Each `Automaton` object has `Location` objects, with `Edge` objects in the locations that each contains a CIF `edge`, all the way down to a `not` operator or a `12` integer value in an `Expression` in (for example) an update of an edge.

All CIF objects that are defined at one place and used elsewhere, such as variables, event declarations, internal user-defined functions (and many others), cross-link from the use back to their definition. These are direct links, that do not follow the tree hierarchy (unlike in CIF text files where the path from use to definition must be stated). The cross-links make it easy to get the definition from its use. They also make it possible to find all uses of a definition without getting confused about uses of a second definition with the same (local) name.

As an example of how to write a CIF file as an object-tree, and what can be found in a CIF object-

tree, consider the following CIF specification:

```
// example.cif

group G1:
end

group G2:
  group H:
    @doc("Controllable event")
    controllable c_event;
  end

  automaton A:
    location:
      initial;
      edge H.c_event;
  end
end
```

To convert this text to a CIF object-tree, a CIF tool must write this specification to a file. The simplest way to do that is to tell the CIF tool that produces this file to write it as an object-tree instead of CIF text by using the `.cifx` extension for its output file. If the CIF specification is already stored, the CIF to CIF transformer can be used, for example with a ToolDef script like:

```
from "lib:cif" import *;

string input_file = "example.cif";
string output_file = "example.cifx"; // <-- Note the ".cifx" here!
cif2cif(
  input_file,
  "--transformations=elim-comp-def-inst,remove-pos-info",
  "--output=" + output_file,
);
```

This script expands all component definitions to their instances, to make the result easier to process. In addition, it removes position information (the line and column numbers of all CIF objects). Generally the latter information is not needed and it avoids a lot of clutter in the output, which is useful if the result is manually inspected. If it is desired to create an object tree file without doing any transformation, remove the `--transformations` option.

Use of the `.cifx` extension causes the CIF file writer to write the CIF object-tree in XMI format, instead of converting it to normal CIF text. XMI is a form of XML, designed to exchange model files (such as CIF models) with `xmi:id` links between element definitions and their uses. The resulting (plain text) file looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<cif:Specification ...>
  <components xmi:type="cif:Group" xmi:id="2" name="G1"/>
  <components xmi:type="cif:Group" xmi:id="3" name="G2">
    <components xmi:type="cif:Group" xmi:id="4" name="H">
      <declarations xmi:type="declarations:Event" xmi:id="5" name="c_event"
controllable="true">
        <annotations xmi:type="annotations:Annotation" xmi:id="6" name="doc">
          <arguments xmi:type="annotations:AnnotationArgument" xmi:id="7">
            <value xmi:type="expressions:StringExpression" xmi:id="8" value=
"Controllable event">
              <type xmi:type="types:StringType" xmi:id="9"/>
            </value>
          </arguments>
        </annotations>
      </declarations>
    </components>
    <components xmi:type="automata:Automaton" xmi:id="10" name="A">
      <locations xmi:type="automata:Location" xmi:id="11">
        <initials xmi:type="expressions:BoolExpression" xmi:id="12" value="true">
          <type xmi:type="types:BoolType" xmi:id="13"/>
        </initials>
        <edges xmi:type="automata:Edge" xmi:id="14">
          <events xmi:type="automata:EdgeEvent" xmi:id="15">
            <event xmi:type="expressions:EventExpression" xmi:id="16" event="5">
              <type xmi:type="types:BoolType" xmi:id="17"/>
            </event>
          </events>
        </edges>
      </locations>
    </components>
  </components>
</cif:Specification>
```

For brevity, a long list of XML declarations at the second line are omitted above. For working with the file however, they are needed.

When comparing the entries in this file with the original CIF specification, it is easy to see how the structure of the CIF specification is reflected in the XMI file. The `Specification` element has two `Group` elements named `G1` and `G2`, just like in the CIF file. In the second group, there is another `Group` element named `H`. The latter group element contains an `Event` named `c_event`, which in turn has a `Annotation` named `doc`. Automaton `A` is the second element in `G2`. It has a `Location` with true `initials` and an `Edge`. The edge has one `EdgeEvent` since it has only one event. The event itself is then stored in an `EventExpression` with an `event="5"` cross-link that corresponds with `xmi:id="5"` of the `c_event` declaration earlier in the file.

For larger CIF specifications, the output file grows quickly in the number of XMI nodes. Each node however carries similar information as above.

Doing a few experiments like above helps in getting an intuition for what is stored in an object-tree. The full CIF language however has many classes, and your experiments may not cover all possible

object-trees. In addition, in some cases the meaning of a class or a data field may not be clear. To better understand the object-trees, consult the extensive documentation that covers all details. Section The CIF meta-model explains where to find the documentation.

Manually tracking all the nodes and connections in an XMI file is tedious work. The obvious next step is thus to have a computer do this for us. That is discussed next.

7.1.3. Getting relevant information from the object-tree

As discussed in the Structure of the CIF object-tree section, it is possible to obtain a `.cifx` file with a CIF object-tree that contains all information of the CIF model. The next step is to load the `.cifx` XML file into Python, and select the desired information from it.

As XMI is a form of XML, you can load an XMI file using an XML library. For Python, the recommended way for loading XML files is to use the ElementTree module.

Loading the `.cifx` file with `ElementTree` is as simple as:

```
import xml.etree.ElementTree as ET

# Define the 'xmi' namespace, needed for finding nodes in the tree.
namespaces = {
    'xmi': 'http://www.omg.org/XMI',
}

# Load the XMI file.
doc = ET.parse('example.cifx')
```

All information from the CIF file is stored in the loaded tree. The next step to get the desired information from the file is to find the nodes in the tree that contain the desired information for your application, and to extract the required data from them. For example, you could extract the names of automata locations, the texts of `@doc` annotations, or the types of discrete variables.

For very small trees, finding nodes in the loaded document can be done ‘manually’ by starting from the root of the tree. Each node is inspected, and depending on the result, nodes deeper in the tree can be considered. Eventually, a node with the desired information may be found. In that case, the relevant information is extracted from it and the search continues for more information.

For the CIF language, with hundreds of different nodes and often a very large tree, that approach would need a lot of Python code, which takes a lot of effort to write and test for correctness. As large XML files are commonly used, the XML community invented the XPath language to easily and efficiently find relevant parts of XML trees. The Python `ElementTree` module also supports that, as described in the XPath support section of the `ElementTree` manual.

XPath takes as input a description of a path to the desired nodes. It then performs the search, and selects and returns the nodes that match the description. The found nodes can then be queried for the relevant information, that can be used in the application.

XPath finds the nodes expressed in the given path by having a set of selected nodes, and updating

that set as it processes each path element. When the entire path is processed, the final set of selected nodes is then returned as result of the search. The Supported XPath syntax table described the details of how each supported path element updates the selected nodes.

As an example, consider the `.//components[@xmi:type="automata:Automaton"]` path:

- The `.` path element selects the current node (at the root of the tree). For the CIF node tree, that is the `Specification` node.
- The `//` path element selects all nodes below the previous selection (so, its children, the children of its children, and so on). All nodes in the tree are selected now, except the root node.
- The `components` path element selects all nodes from the previous selection that can be directly reached by a `components` tag. For the CIF node tree, all `Group` nodes and all `Automaton` nodes are selected. If you have component instantiations (concrete instances of group definitions or automaton definitions), they will be selected as well.
- The `[@xmi:type="automata:Automaton"]` path element restricts the selection to the nodes that have an XMI `type` attribute with the value `automata:Automaton`. For the CIF node tree, the selection now only contains all `Automaton` nodes.

More path elements can be added, thus allowing to find very specific nodes.

Performing the XPath selections in Python takes only a handful of lines. Below are a few queries to get started.

Look at the stated path and the nodes in the loaded `.cifx` file, and compare against the printed results of the Python script. Also check against the CIF classes in the meta-model. Last but not least, try to modify the CIF file or the XPath search and check whether it works as expected.

Find all nodes in a `components` list:

```
# Import ElementTree and set the name spaces as shown above.
doc = ET.parse('example.cifx')

for elem in doc.findall('.//components'):
    print(f"Component {elem.get('name')}")
```

The query produces:

```
Component G1
Component G2
Component H
Component A
```

Find all nodes in a `components` list, and restrict it to those components that are an `automata:Automaton` CIF object:

```
# Import ElementTree and set the name spaces as shown above.
```

```
doc = ET.parse('example.cifx')

for elem in doc.findall('.//components[@xmi:type="automata:Automaton"]', namespaces):
    print(f"Automaton: {elem.get('name')}")
```

The query produces:

```
Automaton: A
```

Find all nodes in a `declarations` list, then restrict them to `declarations:Event` types, then select all nodes in them that are in `annotations` lists, then restrict to `doc` names, and finally select the parent (via `..`) of the matched annotation node to get event declarations as result:

```
# Import ElementTree and set the name spaces as shown above.
doc = ET.parse('example.cifx')

path_spec = ('.//declarations'
           + '[@xmi:type="declarations:Event"]'
           + '/annotations'
           + '[@name="doc"]'
           + '/..')
for elem in doc.findall(path_spec, namespaces):
    print(f"Event with @doc annotation: {elem.get('name')}")
```

The query produces:

```
Event with @doc annotation: c_event
```

Further extraction of specific information from the selected nodes, such as the `elem.get('...')` above is explained in the Tutorial section of the `ElementTree` module documentation.

For constructing new queries, start by writing an example CIF file that is to be queried, check the CIF meta-model information about the tree that can be expected, and create an XPath query expression in an incremental way, rather than trying to construct it completely in one attempt.

7.1.4. Advanced access or modification of a CIF specification

In the previous section it was demonstrated how to extract information from a CIF file in a reliable and efficient way by using an XML library. For many uses that is sufficient.

However, if it is desired to modify CIF objects or even to create entirely new parts in CIF models, that can be more involved when using an XML library. For example, it requires several lines of Python code to create a new `Automaton` object with a number of locations, and add it to the model.

In such cases, it may be easier to use an Ecore library instead of an XML library. An Ecore library allows you to more easily create objects or manipulate them. The ESCET project uses the Eclipse

Modeling Framework (EMF) as Ecore library to work with CIF models. The definition of the CIF objects is available in the `cif/org.eclipse.escet.cif.metamodel/model/cif.ecore` file. The EMF classes are available as well. Using the Java code of the ESCET project to modify CIF models is therefore an option. Other languages may also have Ecore support.

In all cases, the resulting object-tree must comply with the restrictions defined in the CIF meta-model section. Failure to do so may result in undefined behavior by tools that load the resulting tree. The ESCET project performs some validation when loading `.cifx` files.

7.1.5. The CIF meta-model



This section provides a global overview of everything that may be found in a CIF object-tree. At first sight it may seem overwhelming, especially when trying to remember all information. As the information is not going anywhere, the suggestion is to browse through it for some time to understand what kind of information is available. When you have detailed questions about something specific in a CIF object-tree, return here and look into that particular part in more detail.

The CIF meta-model is kept in the `cif/org.eclipse.escet.cif.metamodel` folder in the ESCET Git repository. It is a set of 10 packages. Each package covers a part of the CIF language and contains multiple classes.

To understand what classes exist in the CIF meta-model and how they relate to each other, all classes have been drawn in a UML class diagram. There is one class diagram for each package of the CIF meta-model. The class diagrams are also available as `.png` files. The name of a file corresponds with the name of the package in the CIF meta-model that it depicts:

- `cif/org.eclipse.escet.cif.metamodel/model/images/cif.png`: Structure of the overall specification and components.
- `cif/org.eclipse.escet.cif.metamodel/model/images/automata.png`: Structure of an automaton.
- `cif/org.eclipse.escet.cif.metamodel/model/images/declarations.png`: Declarations of variables, types, events, and so on.
- `cif/org.eclipse.escet.cif.metamodel/model/images/functions.png`: Internal and external user-defined functions.
- `cif/org.eclipse.escet.cif.metamodel/model/images/expressions.png`: Expressions, from literal `true` to `if` expressions.
- `cif/org.eclipse.escet.cif.metamodel/model/images/types.png`: Data types, from booleans to dictionaries and tuples.
- `cif/org.eclipse.escet.cif.metamodel/model/images/print.png`: Print declarations.
- `cif/org.eclipse.escet.cif.metamodel/model/images/cifsvg.png`: CIF/SVG declarations.
- `cif/org.eclipse.escet.cif.metamodel/model/images/annotations.png`: Annotations.
- `common/org.eclipse.escet.common.position.metamodel/model/position.png`: Position information (lines and columns in a CIF file).

The diagrams make extensive use of inheritance, containment, and association. If these concepts are not familiar to you, it may be a good idea to first understand them by reading about UML class diagrams or object-oriented programming.

The diagrams are a good way to get an understanding of how classes relate to each other, but they lack a complete description of the meaning of each field of each class. Those descriptions are available in the `cif/org.eclipse.escet.cif.metamodel/docs/cif_ecore_details.pdf` file. It contains all details about all CIF language constructs. Not something to read from first to last page, but it should provide an answer to any technical detail question about CIF objects and the CIF language.

7.2. CIF language modification

For each change to the CIF language, follow these steps:

- Update textual syntax:
  - Update `cif.setext` syntax (`org.eclipse.escet.cif.parser/src/org/eclipse/escet/cif/parser`).
  - Update `cif.bnf` syntax (`org.eclipse.escet.cif.documentation/images/language-reference/syntax`).
  - Update `cif.min.js` syntax (`org.eclipse.escet.releng.configuration/highlightjs-assets/escet/languages/cif.min.js`).
- Update metamodel (`org.eclipse.escet.cif.metamodel`).
  - Update Ecore diagrams (open `model/cif.aird` and unfold *Design* and then *Entities in a Class Diagram*).
  - Update diagram images.
    - Remove all `*.png` files in the `model/images` folder of the CIF metamodel project.
    - Right click the `cif.aird` file in the *Package Explorer*` and choose **Export representations as images**.
    - Set *To directory* to the `model/images` folder of the CIF metamodel project.
    - Set *Image Format* to *PNG*.
    - Set *Image Size* to *Nominal*.
    - Click **[OK]**.
  - Update genmodel.
    - Delete `model/cif.genmodel`.
    - Right click `model/cif.ecore` and select **New › Other...**.
    - Select *Eclipse Modeling Framework* and *EMF Generator Model* and click **[Next]**.
    - The parent folder and the file name should already be filled, click **[Next]**.
    - Select *Ecore model* and click **[Next]**.
    - Click **[Load]** and then **[Next]**.
    - For *Root packages*, select *cif*, for *Referenced generator models*, select *Position*, and click **[Finish]**.

        - Run `model/autofix.py` to update the newly created `cif.genmodel`.
    - Generate metamodel code.
        - Remove the generated model code (contents of `src-gen`).
        - Open `model/cif.genmodel`.
        - Right click *Cif* and click **Generate Model Code**.
- Update generated Java constructors and walker code (`org.eclipse.escet.cif.metamodel.java`).
    - Remove the generated code (contents of `src-gen`).
    - Right click `gen-cif-constructors.launch` and click **Run As › gen-cif-constructors**.
    - Right click `gen-cif-walker.launch` and click **Run As › gen-cif-walker**.
- Update LaTeX documentation (`org.eclipse.escet.cif.metamodel`).
    - Right click `gen-cif-docs-ecore-latex-skeleton.launch` and click **Run As › gen-cif-docs-ecore-latex-skeleton**.
    - Right click `gen-cif-docs-ecore-latex-skeleton-detail.launch` and click **Run As › gen-cif-docs-ecore-latex-skeleton-detail**.
    - Update `docs/cif_ecore_doc.tex` based on the changes in `docs/cif_ecore_doc_generated.tex`.
    - Update `docs/cif_ecore_doc_details.tex` based on the changes in `docs/cif_ecore_doc_details_generated.tex`.
    - Run `docs/build.bash` to generate a new PDF.
- Update parser (`org.eclipse.escet.cif.parser`).
    - Right click `gen-cif-scanner-parser.launch` and click **Run As › gen-cif-scanner-parser**.
    - Implement new AST classes if needed.
    - Update `CifParserHooks.java` such that it implements `CifParser.Hooks`.
    - Verify the implementation by running `chk.bash`.
- Update keywords and syntax highlighting for the text editor (`org.eclipse.escet.cif.texteditor`).
- Update type checker (`org.eclipse.escet.cif.typechecker`).
    - Add new constraints to the error and problem messages (`ErrMsg.java`).
    - Verify the implementation by running `chk.bash`.
- Update example and benchmark models for changed syntax and constraints (`org.eclipse.escet.cif.models`).
- Update pretty printer including set of keywords (`org.eclipse.escet.cif.prettyprinter`).
- Update common methods (`org.eclipse.escet.cif.common`).
- Update CIF to CIF transformations and other tools.
- Update CIF documentation.
    - Update CIF language reference manual with lexical syntax and grammar.
    - Update CIF developers documentation with the Extracting information from a CIF specification section.

    - A `.cifx` model is shown.
    - It links to the metamodel packages and detailed PDF documentation.
    - Example XML queries are shown and explained that demonstrate getting information from the CIF model tree.
- Update CIF syntax highlighting definition for LaTeX (`org.eclipse.escet.cif.misc/latex/cif_listing.sty`).

Chapter 8. CIF history

This page explains some of the history of CIF, and how its purpose and use have changed over time. It also explains what 'CIF' originally stood for.

8.1. Conception

The CIF language and toolkit were originally created to establish interoperability between a wide range of tools and their associated modeling languages. Languages and tools considered to be in scope at the time were those from computer science and from dynamics and control, and focusing on modeling, simulation, analysis, controller synthesis, and verification in the area of discrete event, timed, and hybrid systems. The various tools, such as UPPAAL and Modelica, had different capabilities. Since each tool had its own associated language, unlocking these capabilities for other languages required model transformations between the various languages.

However, this was considerable work, as for each language a transformation to each other language had to be created. The idea of an interchange format (CIF) was born to reduce the number of transformations that are needed. Instead of transforming n languages each to $n - 1$ others (a quadratic number of transformations), each language would only need to be transformed to and from CIF. For instance, a UPPAAL model could be transformed to CIF, and then to a Modelica model. That way, only $2 * n$ transformations would be needed (a linear number of transformations), reducing the effort to implement the transformations.

The interchange format needed to be rich enough to represent the concepts from the various languages. Only then could a large enough subset of language features of the various languages be transformed. Hence, CIF needed concepts from discrete event, timed, and hybrid systems.

In order for the transformations to be useful, they had to be semantically preserving. The model in one language, when transformed to another, should still behave in the same way. That way, the analysis results would also be valid for the original model, allowing different tools to be combined in a consistent and meaningful way. To enable this, it was deemed necessary for CIF to have a formal semantics, grounded in mathematics, and independent of any implementation aspects.

Different languages may have different concepts. A direct transformation from a source language to a destination language would therefore have to deal with all this complexity. Instead, as a pre-transformation step, the source model could be adapted such that concepts that only exist in the source language are transformed to equivalent concepts in the source language that also exist in the destination language. This idea was considered useful for many of the language-to-language transformations. By doing such pre-transformations on the interchange format itself, by using CIF to CIF transformations to rewrite CIF models, they could be reused rather than having to implement them again for each language-to-language transformation. For such pre-transformations to be valid, it is essential that parts of a model that are not touched keep their meaning. This property for a (interchange) language is known as the principle of compositionality. The semantics of the CIF language thus needed to be compositional.

There was thus a need for an interchange format with formal compositional semantics that supports concepts from discrete event, timed, and hybrid systems. For this, the '**C**ompositional **I**nterchange **F**ormat for hybrid systems', abbreviated as 'CIF', was created. In CIF, hybrid systems

conceptually included timed systems, and timed systems conceptually included discrete event systems. Hence, only hybrid systems were mentioned in the full name.

8.2. CIF 1

The first version of the CIF language, CIF 1, was defined in 2006. The first CIF tools were created in 2007, in Python. CIF was at that time essentially a front-end language. The first tool, the CIF simulator, represented CIF models internally as Chi models. The CIF 1 toolset was later extended with other tools, including various transformations to other languages and support for supervisory controller synthesis. Along with CIF 1, the Supervisory Control IDE (SCIDE), a graphical editor for supervisory control models, was created.

The work around CIF 1 was primarily carried out in several academic research projects, including the EU NoE project HYCON, the EU ITEA2 project Twins, and the Darwin project. The CIF 1 tooling was in development from 2007 to around 2012.

A few years after its inception, several issues were identified with CIF 1. Its language concepts were difficult to use, the textual syntax was cumbersome, and the semantics was hard to get right. The tooling was slow and ran only on Linux. The ad-hoc architecture, built on top of Chi, was hard to maintain, and had a considerable number of bugs that were non-trivial to solve.

Here are some publications about CIF 1:

- D. A. van Beek, M. A. Reniers, R. R. H. Schiffelers, and J. E. Rooda, "Foundations of a compositional interchange format for hybrid systems", Computer science reports, volume 633, Eindhoven University of Technology.
- D. A. van Beek, M. A. Reniers, R. R. H. Schiffelers, and J. E. Rooda, "Foundations of a Compositional Interchange Format for Hybrid Systems", Hybrid Systems: Computation and Control (HSCC), pages 587-600, 2007, doi:10.1007/978-3-540-71493-4_45.
- D. A. van Beek, M. A. Reniers, J. E. Rooda, and R. R. H. Schiffelers, "Concrete syntax and semantics of the compositional interchange format for hybrid systems", IFAC Proceedings Volumes, volume 41, issue 2, pages 7979-7986, 2008, doi:10.3182/20080706-5-KR-1001.01348.
- C. Sonntag, R. R. H. Schiffelers, D. A. van Beek, J. E. Rooda, and S. Engell, "Modeling and Simulation using the Compositional Interchange Format for Hybrid Systems", International Conference on Mathematical Modelling (MATHMOD), pages 640-650, 2009.
- D. A. van Beek, P. Collins, D. E. Nadales, J. E. Rooda, and R. R. H. Schiffelers, "New Concepts in the Abstract Format of the Compositional Interchange Format", IFAC Proceedings Volumes, volume 42, issue 17, pages 250-255, 2009, doi:10.3182/20090916-3-ES-3003.00044.

8.3. CIF 2

To solve the difficulties with CIF 1, it was decided to start from scratch, resulting in CIF 2. The focus of CIF 2 was on selecting the right concepts and getting their semantics right.

The first release of the CIF 2 tooling was on September 8, 2010. With CIF 2, CIF remained an interchange format with various transformations to other languages. Transformations were supported by a model transformation framework based on the Eclipse Modeling Framework (EMF).

It allowed for transformations to and from the CIF 2 metamodel, a standard representation of CIF models. Transformations could be implemented using model transformation languages such as ATL and QVTo, rather than general purpose languages such as Java.

The work around CIF 2 was primarily carried out in several academic research projects, including the Darwin project, the EU FP7 project C4C, the EU FP7 project Multiform, and the EU NoE project HYCON 2. The CIF 2 tooling was in development from 2010 to around 2014.

Unfortunately, the role of CIF 2 as an interchange format was never fully realized. The concepts of the different languages often turned out to be quite different, or appeared similar but had different semantics. To compensate for these differences, transformations 'encoded' concepts in terms of other concepts. Similar to a dictionary that explains an unknown word in terms of many known words, transformations used elaborate combinations of CIF concepts to express the foreign concept in CIF. This made it difficult to subsequently transform such CIF models to other languages. After a while, interest in CIF as an interchange format declined.

Besides that, the CIF 2 syntax was not very user friendly. Furthermore, the implementation technology around metamodels and model transformation languages was not very mature at the time. The CIF 2 tooling was difficult to use and difficult to maintain.

Here are some publications about CIF 2:

- D. E. Nadales Agut, M. A. Reniers, R. R. H. Schiffelers, K. Y. Jørgensen, and D. A. van Beek, "A Semantic-Preserving Transformation from the Compositional Interchange Format to UPPAAL", IFAC Proceedings Volumes, volume 44, issue 1, pages 12496-12502, 2011, doi:10.3182/20110828-6-IT-1002.03030.
- D. Hendriks, R. R. H. Schiffelers, M. Hüfner, C. Sonntag, "A Transformation Framework for the Compositional Interchange Format for Hybrid Systems", IFAC Proceedings Volumes, volume 44, issue 1, pages 12509-12514, 2011, doi:10.3182/20110828-6-IT-1002.03561.
- C. Sonntag and M. Hüfner, "On the Connection of Equation- and Automata-based Languages: Transforming the Compositional Interchange Format to Modelica", IFAC Proceedings Volumes, volume 44, issue 1, pages 12515-12520, 2011, doi:10.3182/20110828-6-IT-1002.03565.
- D. E. Nadales Agut, D. A. van Beek, and J. E. Rooda, "Syntax and semantics of the compositional interchange format for hybrid systems", The Journal of Logic and Algebraic Programming (JLAP), volume 82, issue 1, pages 1-52, 2013, doi:10.1016/j.jlap.2012.07.001.

8.4. CIF 3

Not being satisfied with CIF 2, the idea came to develop CIF 3. Work on CIF 3 started in the academic research project HYCON 2, in 2012. CIF 3 reused most of the semantics of CIF 2. The CIF 3 tooling was implemented using proven technology, with Java as the main implementation language. Work on CIF 3 initially focused on a convenient modeling syntax. The first version of CIF 3 was released on January 31, 2013. Later, the work shifted towards tooling to support the entire development process of supervisory controllers.

With CIF 3, the use of CIF as an interchange format was completely dropped, although CIF 3 featured several transformations to other languages. Over time, CIF 3 started to develop as a language of its own. While the name 'CIF' stuck, it no longer stood for 'Compositional Interchange

Format'.

CIF 3 has been developed from the start with the aim to have industrial use. Hence, considerable effort was put into a good user experience, including good performance, robustness, intuitive and user-friendly error messages, and extensive documentation. CIF 3 has been applied in many academic research projects together with industry. However, even with these considerable efforts, CIF 3 remained an academic tool lacking wide-spread adoption in industry.

Here are some publications about CIF 3:

- D. A. van Beek, W. J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. M. van de Mortel-Fronczak, and M. A. Reniers, "CIF 3: Model-Based Engineering of Supervisory Controllers", Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pages 575-580, 2014, doi:10.1007/978-3-642-54862-8_48.
- W. J. Fokkink, M. A. Goorden, J. M. van de Mortel-Fronczak, F. F. H. Reijnen, and J. E. Rooda, "Supervisor Synthesis: Bridging Theory and Practice", Computer, volume 55, number 10, pages 48-54, 2022, doi:10.1109/MC.2021.3134934.

8.5. Eclipse ESCET

Versions 1 through 3 of CIF were developed by the Eindhoven University of Technology (TU/e) in the Netherlands. In 2018, the idea came to move CIF to the Eclipse Foundation. The goal was, together with other interested parties, to create a larger community around CIF. The aim was to make it easier for all parties in the ecosystem to contribute together to the development of CIF in an open setting.

In September 2019, the proposal for the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) project, which includes CIF, was submitted to the Eclipse Foundation. The proposal was supported by TU/e, TNO, Rijkswaterstaat, and ASML. The Eclipse Foundation's board of directors approved the creation of the project under the MIT license, on November 13, 2019. After a successful creation review, the project was formally created on January 15, 2020. The initial contribution to the project, which included CIF 3, was formally approved on October 16, 2020. The version number for CIF was dropped at that time. The first release of the Eclipse ESCET project, including CIF, was on April 2, 2021. Following a formal graduation review, the project became a mature Eclipse Foundation open-source project, on September 11, 2023.

With the move to the Eclipse Foundation, and supported by the Eclipse Foundation's principles of transparency, openness, meritocracy and vendor-neutrality, the ESCET project is now an open ecosystem with a growing community. Interested parties, such as academic and applied research institutes, industrial partners, and tool vendors, collaborate on and profit from further tool development for the model-based construction of supervisors. Together they are developing the ESCET toolkit as an industrial-strength toolkit.

Here are some publications about Eclipse ESCET:

- W. J. Fokkink, M. A. Goorden, D. Hendriks, D. A. van Beek, A. T. Hofkamp, F. F. H. Reijnen, L. F. P. Etman, L. Moormann, J. M. van de Mortel-Fronczak, M. A. Reniers, J. E. Rooda, L. J. van der Sanden, R. R. H. Schiffelers, S. B. Thuijsman, J. J. Verbakel, and J. A. Vogel, "Eclipse ESCET™: The Eclipse Supervisory Control Engineering Toolkit", Tools and Algorithms for the Construction and

Analysis of Systems (TACAS), pages 44-52, 2023, doi:10.1007/978-3-031-30820-8_6.

- D. Hendriks, M. A. Reniers, W. J. Fokkink, and W. H. M. Oortwijn, "Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0", arXiv, 2025, doi:10.48550/arXiv.2511.04370.

Chapter 9. Legal

The material in this documentation is Copyright (c) 2010, 2026 Contributors to the Eclipse Foundation.

Eclipse ESCET and ESCET are trademarks of the Eclipse Foundation. Eclipse, and the Eclipse Logo are registered trademarks of the Eclipse Foundation. Other names may be trademarks of their respective owners.

License

The Eclipse Foundation makes available all content in this document ("Content"). Unless otherwise indicated below, the Content is provided to you under the terms and conditions of the MIT License. A copy of the MIT License is available at https://opensource.org/licenses/MIT. For purposes of the MIT License, "Software" will mean the Content.

If you did not receive this Content directly from the Eclipse Foundation, the Content is being redistributed by another party ("Redistributor") and different terms and conditions may apply to your use of any object code in the Content. Check the Redistributor's license that was provided with the Content. If no such license exists, contact the Redistributor. Unless otherwise indicated below, the terms and conditions of the MIT License still apply to any source code in the Content and such source code may be obtained at https://www.eclipse.org.

Third Party Content

The Content includes items that have been sourced from third parties as set out below. If you did not receive this Content directly from the Eclipse Foundation, the following is provided for informational purposes only, and you should look to the Redistributor's license for terms and conditions of use.

- *Font Awesome*

  The Content includes parts of Font Awesome. Font Awesome is licensed under the SIL Open Font License 1.1.

  SPDX-License-Identifier: OFL-1.1

- *Highlight.js*

  The Content includes parts of highlight.js. Highlight.js is licensed under the BSD 3-Clause License.

  SPDX-License-Identifier: BSD-3-Clause

- *MathJax*

  The Content includes parts of MathJax. MathJax is licensed under the Apache License, Version 2.

  SPDX-License-Identifier: Apache-2.0

- *Wikipedia external link icon*

The Content includes the Wikipedia external link icon. The external link icon is licensed under the Creative Commons Attribution 4.0 International license.

SPDX-License-Identifier: CC-BY-4.0

Index

@

!, 133
    channel parameter usage restriction, 149
$, 229
., 229, 230
?, 133
    channel parameter usage restriction, 149
^, 230
~
    channel parameter usage restriction, 149

A

actuator
    digital, 30
add default initial values, 604
    implementation, 604
    optimality, 605
    preprocessing, 604
    renaming, 605
    size considerations, 605
    supported specifications, 604
addressable, 78
algebraic variable, 116
    eliminate, 619
    equation, 118
algebraic variables
    remove unused, 704
alphabet, 62
    default, 62
    explicit, 63
    global event disablement, 64
    implicit, 62
    implicit vs explicit, 64
    minimum, 64
    non-default, 64
annotation, 217, 286
    controller, 218
    doc, 218
    name, 230
annotations
    remove, 697
anonymize names
    implementation, 606
    optimality, 608
    preprocessing, 606
    renaming, 608
    size considerations, 608
    supported specifications, 605
array, 109
assignment, 69
    addressable, 78
    left hand side, 78
    multi-assignment, 79
    multiple, 78
    order, 79
    right hand side, 78
    value, 78
automata to string casts
    eliminate, 621
automatic input mode, 410
    automatic mode choice algorithm option, 410
    automatic mode transition duration option, 411
    limitations, 412
automaton, 56
    alphabet, 62
    definition, 144
    instantiation, 144
    kindless, 213
    monitor, 77
    plant, 213
    regular, 213
    requirement, 213
    supervisor, 213
    supervisory kind, 213
automaton definition, 144
    parameters, 146, 147
automaton definition parameter, 146
    algebraic, 147
    automaton, 152
    channel, 149
    component, 152
    event, 148
    input, 151
    location, 150
    multiple, 154
automaton instantiation, 144
    arguments, 146, 147
automaton self reference
    eliminate, 641

B

boolean, 102
  literal, 102
  type, 102
  value, 102

C

C89
  code generation, 520
C99
  code generation, 526
channel, 130
  dataless, 134
  declaration, 133
  receive, 133
  received value, 133
  receiver, 132
  send, 133
  sender, 132
  synchronization, 135
  void, 134
channel parameter usage restriction
  !, 149
  ?, 149
  flags, 149
  ~, 149
CIF code generator, 514
  C89, 520
  C99, 526
  HTML, 544
  Java, 532
  JavaScript, 538
  options, 515
  preprocessing, 517
  Simulink, 549
  start, 515
  supported specifications, 516
CIF event disabler, 725
  options, 726
  output, 727
  performance, 728
  preprocessing, 726
  specifying events, 727
  start, 725
  supported specifications, 726
  usage scenario, 725
CIF explorer, 728
  options, 729
  preprocessing, 731
  start, 729
  supported specifications, 730
CIF information generator, 300
CIF merger, 716
  example, 718
  merge compatibility, 721
  merge problems, 724
  options, 717
  preprocessing, 718
  shared events, 721
  shared variables, 721
  start, 716
CIF multi-level splitter, 360
  options, 362
  output, 365
  preprocessing, 363
  start, 362
  supported specifications, 363
CIF PLC code generator, 582
  accuracy over time, 589
  generated code, 586
  I/O coupling, 589
  IEC 61131-3, 590
  names, 587
  options, 582
  output types, 590
  PLCopen XML, 590
  preprocessing, 585
  program body, 588
  Siemens S7 standard compliance, 579
  standard compliance, 586
  start, 582
  supported specifications, 584
  TwinCAT, 590
  TwinCAT standard compliance, 574
CIF relations collector, 302
  options, 302
  output, 304
  preprocessing, 303
  start, 302
CIF simulator
  complete mode, 472
  console output, 427
  deadlock, 394
  debug generated code, 478
  debug output, 432

execution mode, 471
external user-defined functions, 473
fixed output step size, 467
history, 419
initialization, 394
input components, 403
input modes, 403
Java compiler, 470
maximum delay, 467
maximum time point tolerance, 477
non-urgent events, 422
normal output, 428
ODE solver, 458
options, 454
output, 427
output components, 427
performance, 456
plot visualizer, 444
print output, 433
profiling, 476
real-time simulation, 451
repeated simulation, 400
reset, 419
runtime error, 395
simulation end time, 395
simulation of traces, 396
start, 393
starting with generated code, 479
state visualizer, 449
stochastic distributions, 477
supported specifications, 393
SVG visualization, 435
termination, 394
test mode, 480
trajectory data, 441
undo, 419
CIF to CIF transformer, 600
add default initial values, 604
anonymize names, 605
available transformations, 602
convert controllable events to uncontrollable, 615
convert switch expressions to if expressions, 617
convert to interface, 608
convert uncontrollable events to controllable, 614
eliminate algebraic variables, 619
eliminate automata to string casts, 621
eliminate automaton self references, 641
eliminate channels, 623
eliminate component definition/instantiation, 626
eliminate constants, 628
eliminate equations, 630
eliminate groups, 632
eliminate if updates, 635
eliminate monitors, 639
eliminate state invariants, 646
eliminate state/event exclusion invariants, 642
eliminate tau event, 650
eliminate the use of locations in expressions, 637
eliminate tuple field projections, 651
eliminate type declarations, 652
enumerations to constants, 654
enumerations to integers, 655
lift events, 657
linearize (merge), 659
linearize (product), 669, 677
merge enumerations, 688
options, 601
pretty printer, 603
push print file declarations into print declarations, 692
push SVG file declarations into other CIF/SVG declarations, 694
relabel supervisors as plants, 696
remove annotations, 697
remove CIF/SVG declarations, 698
remove I/O declarations, 699
remove position information, 701
remove print declarations, 700
remove requirements, 702
remove unused algebraic variables, 704
remove unused events, 706
simplify others, 708
simplify values, 709
simplify values (no references), 713
simplify values (no references, optimized), 715
simplify values (optimized), 712
start, 600
transformation descriptions, 601
CIF to mCRL2 transformer, 480

options, 481
preprocessing, 483
result of the transformation, 484
start, 480
supported specifications, 481
CIF to Supremica transformer, 386
alphabets, 390
initialization, 389
jumping semantics, 391
marking, 390
options, 387
preprocessing, 388
range semantics, 390
start, 386
state invariants, 390
supported specifications, 387
transformation result, 389
CIF to UPPAAL transformer, 488
options, 488
preprocessing, 490
start, 488
supported specifications, 489
transformation result, 490
CIF to yEd transformer, 731
diagram kinds, 733
layout, 739
options, 732
preprocessing, 733
start, 731
supported specifications, 732
CIF/SVG declaration, 174, 248
remove, 698
code
generation, 13
comments, 232
multi line, 232
single line, 232
component, 4, 56, 156
automaton, 56
automaton instantiation, 144
definition, 156
group, 154
group instantiation, 155
mechanical, 4
component definition, 144
eliminate, 626
component instantiation, 144
eliminate, 626
constant, 115
eliminate, 628
expression, 98
value, 98
continuous variable, 122
derivative, 123
equation, 124
control
conditions, 19
requirement, 14
software, 14
controllable, 19
controller, 19, 20, 22
annotation, 218
design, 6
model, 10, 14
realization, 6
reliability, 20
resource, 4
supervisor, 19
supervisory, 3
synthesis, 19
validation, 14
verification, 14
controller properties checker, 491
execution scheme, 499
options, 493
preprocessing, 498
runtime errors, 501
start, 493
supported specifications, 493
convert controllable events to uncontrollable, 615
implementation, 616
preprocessing, 616
supported specifications, 616
convert switch expressions to if expressions, 617
implementation, 617
optimality, 619
size considerations, 618
convert to interface, 608
example, 608
implementation, 612
preprocessing, 611
supported specifications, 611
convert uncontrollable events to controllable, 614
implementation, 614

preprocessing, 614
supported specifications, 614
correct-by-construction, 12, 16, 20
counter example, 16

D

data-based supervisory controller synthesis, 306
checks, 316
debugging, 353
edge granularity, 328
edge order, 330
fixed-point order, 322
forward reachability, 321
initialization, 324
input variables, 324
operation cache, 351
options, 307
performance, 325
preprocessing, 314
requirements, 315
resulting supervisor, 317
simplification, 319
start, 307
statistics, 355
supported specifications, 311
variable order, 333
deadlock, 129
debug
generated code, 478
maximum delay, 467
PLC code, 589
print output, 201
simulator debug console output, 432
SVG input, 417
SVG visualization, 438
debug console output
option, 432
declaration
multiple, 67
order, 84
derivative, 123
continuous variable, 123
equation, 124
design
computer-aided, 20
freedom, 20
modular, 21
development
incremental, 36
dictionary, 110
literal, 110
pair, 110
type, 110
value, 110
discrete event system, 49
discrete variable, 69
initial value, 83
value change, 69
do, 69
doc
annotation, 218
documentation
reference, 226
DSL, 11, 14

E

edge
assignment, 69
do, 69
guard, 69
multiple events, 67
no events, 83
self loop, 66, 76
take, 57
update, 69
when, 69
edge goto, 57
eliminate algebraic variables, 619
implementation, 619
optimality, 620
preprocessing, 619
size considerations, 620
supported specifications, 619
eliminate automata to string casts, 621
implementation, 621
optimality, 622
preprocessing, 621
size considerations, 622
supported specifications, 621
eliminate automaton self references, 641
implementation, 642
preprocessing, 641
size considerations, 642
supported specifications, 641
eliminate channels, 623
eliminate component definition/instantiation,

626
implementation, 627
size considerations, 628
eliminate constants, 628
implementation, 629
optimality, 629
preprocessing, 629
size considerations, 629
supported specifications, 628
eliminate equations, 630
implementation, 630
optimality, 631
preprocessing, 630
size considerations, 631
supported specifications, 630
eliminate groups, 632
implementation, 632
preprocessing, 632
renaming, 633
size considerations, 635
supported specifications, 632
eliminate if updates, 635
implementation, 636
optimality, 637
preprocessing, 635
size considerations, 637
supported specifications, 635
eliminate monitors, 639
implementation, 640
optimality, 641
preprocessing, 640
size considerations, 641
supported specifications, 639
eliminate state invariants, 646
implementation, 647
optimality, 649
preprocessing, 647
renaming, 649
size considerations, 649
supported specifications, 646
eliminate state/event exclusion invariants, 642
implementation, 643
optimality, 646
preprocessing, 643
renaming, 645
size considerations, 645
supported specifications, 643
eliminate tau event, 650
implementation, 650
renaming, 651
size considerations, 651
eliminate the use of locations in expressions, 637
implementation, 637
optimality, 639
preprocessing, 637
renaming, 639
size considerations, 639
supported specifications, 637
eliminate tuple field projections, 651
implementation, 651
size considerations, 652
eliminate type declarations, 652
implementation, 653
preprocessing, 652
size considerations, 653
supported transformations, 652
engineering
incremental, 21
model-based, 6, 10, 15
model-driven, 6, 10
synthesis-based, 2, 7, 16, 22
traditional, 6, 7
verification-based, 7, 15
enumeration, 102
convert to consts, 654
convert to ints, 655
declaration, 102
literal, 102
merge, 688
type, 102
value, 102
enumerations to constants, 654
implementation, 654
preprocessing, 654
renaming, 655
size considerations, 655
supported specifications, 654
enumerations to integers, 655
implementation, 656
preprocessing, 656
renaming, 657
size considerations, 657
supported specifications, 655
equation
algebraic variable, 118

benefit, 126
continuous variable, 124
derivative, 124
eliminate, 630
event, 18, 22, 57
channel, 130
controllability, 215
controllable, 18, 21, 215
declaration, 57, 67
determine, 29
disabled, 59
enabled, 59
enabledness, 59
input, 415
interactive, 415
lift to specification, 657
model, 29
multiple on edge, 67
placement, 64
state/event exclusion invariant, 94
synchronize, 58
tau, 81
uncontrollable, 18, 23, 215
urgency, 128, 422
event-based automaton abstraction, 384
event-based synthesis toolset, 384
options, 384
start, 384
event-based automaton projection, 382
event-based synthesis toolset, 382
options, 382
start, 382
event-based controllability check, 378
event-based synthesis toolset, 378
options, 379
start, 378
supported specifications, 378
event-based DFA minimization, 381
event-based synthesis toolset, 381
start, 381
event-based language equivalence check, 379
event-based synthesis toolset, 379
options, 380, 381
start, 379
event-based NFA to DFA automaton conversion, 380
event-based synthesis toolset, 380
options, 380
start, 380
event-based nonconflicting check, 377
event-based synthesis toolset, 377
options, 378
start, 377
event-based observer check, 383
event-based synthesis toolset, 383
options, 383
start, 383
event-based supervisor synthesis, 371
event-based synthesis toolset, 371
options, 372
start, 372
supported specifications, 371
event-based synchronous product, 370
event-based synthesis toolset, 370
options, 370
start, 370
event-based synthesis analysis, 373
event-based synthesis analysis toolset, 373
options, 377
start, 376
event-based synthesis analysis toolset
event-based synthesis analysis, 373
event-based synthesis toolset, 367
automaton kinds, 368
event-based automaton abstraction, 384
event-based automaton projection, 382
event-based controllability check, 378
event-based DFA minimization, 381
event-based language equivalence check, 379
event-based NFA to DFA automaton conversion, 380
event-based nonconflicting check, 377
event-based observer check, 383
event-based supervisor synthesis, 371
event-based synchronous product, 370
event-based trim, 385
event-based trim check, 385
names in errors, 368
names in reports, 368
preprocessing, 368
supported specifications, 367
tools, 369
event-based trim, 385
event-based synthesis toolset, 385
options, 386
start, 386

event-based trim check, 385
  event-based synthesis toolset, 385
  options, 385
  start, 385
events
  convert controllable events to uncontrollable, 615
  convert uncontrollable events to controllable, 614
  remove unused, 706
examples, 747
execution simulation trace, 59
expression, 98
  computation, 98
  constant, 98
  evaluation, 98
  if, 113
  literal, 98
  switch, 114
external user-defined functions, 473
  call by value, 473
  determinism, 473
  Java, 474
  logging, 473
  responsiveness, 473
  side effects, 473
  termination, 473
  value semantics, 473
external user-defined Java functions
  class loader, 475, 476
  implementation reference, 474
  Java Virtual Machine, 475, 476
  lazy loading, 475
  method invocation, 476
  method modifier, 475
  method resolution, 475
  runtime errors, 476
  side effects, 476
  state, 476
  static data, 476
  static method, 475
  stderr, 476
  stdout, 476
  supported types, 474
  supported values, 474

F

fixed point, 322
flags
  channel parameter usage restriction, 149
format specifier, 276
  %B, 279
  %b, 279
  %d, 279
  %E, 281
  %e, 281
  %f, 282
  %G, 283
  %g, 283
  %S, 284
  %s, 284
  %X, 280
  %x, 280
  alignment, 277
  explicit indexing, 277
  flags, 277
  grouping, 277
  implicit indexing, 277
  indexing, 277
  justification, 277
  leading space, 277
  padding, 277
  sign, 277
  syntax, 276
  thousand separator, 277
  width, 278
  zero padding, 277
frame, 451
  intermediate, 452
function, 137
  application, 138
  body, 138
  call, 138
  function variables, 138
  internal, 137
  internal vs external, 137
  local variables, 138
  parameters, 138
  return type, 138
  return value, 138
  side effects, 139
  standard library, 137
  statements, 138, 139
  user-defined, 137
function statements
  assignment, 139

break, 141
continue, 142
if, 140
return, 139
while, 140

G

grammar, 228, 233
group, 154
definition, 155
eliminate, 632
instantiation, 155
namespace, 164
guard, 69
multiple, 465
splitting, 465

H

HTML
code generation, 544
hybrid system, 49

I

I/O declaration
print declaration, 266
print file declaration, 272
remove, 699
SVG copy declaration, 251
SVG declaration, 174, 248
SVG file declaration, 248
SVG input mapping, 264
SVG move declaration, 255
SVG output mapping, 258
identifiers, 229
if, 113
expression, 113
function statement, 140
if update
eliminate, 635
import, 159
directories, 161
group merging, 161
library, 160
initial value
any, 84
default, 83
discrete variable, 83
explicit, 83
implicit, 83
multiple, 84
initialization
consistency, 84, 86
cyclic dependencies, 85
predicate, 85
restricted, 87
Inkscape, 176
changing ids, 177
coordinate system, 178
document size, 178
height, 178
text area, 178
width, 178
XML editor, 178
input mode, 403
automatic input mode, 410
interactive console input mode, 404
interactive GUI input mode, 406
overview, 403
semi-automatic mode, 418
SVG input mode, 415
trace input mode, 413
input variable, 165
co-simulation example, 166
hardware example, 165
merging CIF models, 167
SVG input mapping, 168
int
literal, 231
integer, 99
limits, 100
literal, 99
overflow, 100
range, 100
type, 99
value, 99
integrator options, 468
algorithm, 468
number of steps, 469
step sizes, 468
tolerances, 468
interactive console input mode, 404
semi-automatic mode, 406
interactive GUI input mode, 406
semi-automatic mode, 409
invariant, 91, 94
eliminate state invariants, 646

eliminate state/event exclusion invariants, 642
kindless, 214
plant, 214
regular, 214
requirement, 214
state, 91
state/event exclusion, 94
supervisor, 214
supervisory kind, 214

J

Java
code generation, 532
JavaScript
code generation, 538

K

keyword
escaping, 229
keywords, 228

L

language
domain-specific, 11, 11, 14
formal, 11
modeling, 15
programming, 11, 11, 13
legal, 820
lift events, 657
implementation, 658
preprocessing, 657
renaming, 658
size considerations, 659
supported specifications, 657
linearize
communication, 667, 676, 687
implementation, 660, 669, 678
merge, 659
non-determinism, 662, 672, 681
optimality, 668, 677, 688
order, 664, 673, 684
preprocessing, 659, 669, 678
product, 669, 677
received values, 666, 675, 686
renaming, 667, 676, 687
size considerations, 667, 676, 687
supported specifications, 659, 669, 678
tuple field projections, 666, 675, 686
urgency, 664, 673, 684
list, 105
bounded, 109
comprehension, 107
concatenation, 107
indexing, 105
limits, 108
literal, 105
maximum size, 108
projection, 105
regular, 105
slicing, 106
type, 105
value, 105
literal, 98
boolean, 102
dictionary, 110
enumeration, 102
expression, 98
integer, 99
list, 105
real, 100
set, 109
string, 102
tuple, 103
value, 98
livelock, 130
liveness, 19
location, 57
as variable, 89
initial, 85
invariant, 94
nameless, 68
urgency, 128
versus variable, 70, 71
location initial, 57

M

marker
predicate, 216
marking, 34, 216
merge enumerations, 688
default initial values, 690
implementation, 689
preprocessing, 688
renaming, 689
size considerations, 691

supported specifications, 688
switch expressions, 691
method
correct-by-construction, 22
formal, 10, 14
model, 15, 56
simulation, 44
single source of truth, 13
model-based testing, 12
monitor, 77
eliminate, 639
monitoring, 74
monitor automaton, 77
self loops, 76

N

name
anonymize, 605
names, 229
namespace, 164
non-blocking, 19
non-determinism, 61
non-urgent events
input modes, 423
supervisory control application domain, 424
normal console output, 428
option, 430
output details, 430
state filtering, 431
number
literal, 231

O

ODE, 458
bisection, 463
bracketing, 463
discontinuity, 460
fixed output step size, 467
guard splitting, 465
integration, 458
integrator options, 468
interval, 463
IVP, 458
linear ODE, 458
maximum check interval, 464
maximum delay, 467
nonlinear ODE, 459
numeric integration, 458
numeric root finding, 461
root finder options, 469
root finding, 461
root finding problem, 464
simulation end time, 467
step size, 458
output
print, 201
output component
real-time, 453

P

performance
CIF event disabler, 728
compile once, 400
integrator step sizes, 469
Java compiler, 470
position information, 701
repeated simulation, 400
root finding maximum check interval, 465
simulation, 456
simulation frame rate, 452
SVG input, 417
permissive
maximally, 19, 20
plant, 17, 23, 212
modeling, 29
relations, 30
PLC code generator, 556
@doc, 567
continuous behavior, 567
doc annotations, 567
generated code, 570
I/O table, 564
inputs, 564
names in the PLC code, 571
options, 557
outputs, 564
PLC code names, 571
resulting PLC code, 570
start, 557
supported specifications, 561
timers, 567
plot visualizer, 444
data points, 447
disable, 445
enable, 445
filtering, 445

modes, 446
multiple, 446
range, 447
reset, 449
save plot image, 448
trajectory data relation, 448
undo, 449
print
declaration, 266
output, 201
print declaration, 266
combining filters, 271
escaping, 269
extended trace, 267
file, 272
post text, 268
pre text, 268
push, 692
quoting, 269
remove, 700
scoping, 272
state filtering, 270
stderr, 272
stdout, 272
target, 272
text to print, 268
transition centric view, 266
transition filtering, 269
virtual labels, 267
print file declaration, 272
path, 273
push, 692
remove, 700
scoping, 273
special target, 273
stderr, 273
stdout, 273
uniqueness, 274
unused, 434
print output
disable, 434
enable, 434
example, 201
examples, 202
footer, 206
header, 206
line-based, 434
order, 434
overwrite, 433
print declaration, 266
print file declaration, 272
printing to file, 203
reset, 434
simulation, 433
single open, 433
text formatting, 208
undo, 434
profiling, 476
debug console output, 433
normal console output, 431
push print file declarations into print declarations, 692
implementation, 692
preprocessing, 692
size considerations, 694
supported specifications, 692
push SVG file declarations into other CIF/SVG declarations, 694
implementation, 694
preprocessing, 694
size considerations, 696
supported specifications, 694

R

reachability, 322
backward, 322
forward, 321
real, 100
literal, 100, 231
type, 100
value, 100
real-time simulation, 451
disable, 453
enable, 453
frame, 451
frame rate, 452
intermediate frame, 452
intermediate state, 452
model time delta, 453
simulation speed, 453
test mode, 454
trajectory data, 467
relabel supervisors as plants, 696
implementation, 696
release
notes, 748

remove annotations, 697
implementation, 698
size considerations, 698
remove CIF/SVG declarations, 698
implementation, 699
size considerations, 699
remove I/O declarations, 699
implementation, 699
size considerations, 700
remove position information, 701
implementation, 701
size considerations, 702
remove print declarations, 700
implementation, 700
size considerations, 701
remove requirements, 702
remove unused algebraic variables, 704
remove unused events, 706
requirement, 17, 212
cif, 18
conflicting, 20
design, 5
extra-functional, 5
fifo, 42
functional, 5, 10, 18
mathematical formula, 18
modeling, 32
natural language, 18
remove, 702
safety, 5, 10, 18
traceability, 21
restrictive
minimally, 19, 20
root finder options, 469
algorithm, 469
false position, 469
Illinois, 469
maximum check interval, 469
maximum iterations, 470
Pegasus, 469
Regula Falsi, 469
tolerances, 469

S

S7, 578, 597
generated output, 578, 597
usage, 579, 598
safe, 19
scripting, 299
see: CIF PLC code generator
S7, 578, 597
TwinCAT, 572, 591
see: event-based supervisory control
event-based synthesis toolset, 367
see: explorer
CIF explorer, 728
see: input component
input mode, 403
see: mCRL2
CIF to mCRL2 transformer, 480
see: observing
monitoring, 74
see: ODE solver
ODE, 458
see: output components
simulation output, 427
see: PLC
CIF PLC code generator, 582
PLC code generator, 556
see: pretty printer
CIF to CIF transformer, 603
see: printfile
print file declaration, 272
see: profiling
CIF simulator, 476
see: Supremica
CIF to Supremica transformer, 386
see: SVG visualizer
SVG viewer, 440
SVG visualization, 440
see: svgcopy
SVG copy declaration, 251
see: svgin
SVG input mapping, 264
see: svgmove
SVG move declaration, 255
see: svgout
SVG output mapping, 258
see: trajectory data
ODE, 458
see: UPPAAL
CIF to UPPAAL transformer, 488
see: yEd
CIF to yEd transformer, 731
seed
automatic mode choice algorithm option, 411

automatic mode transition duration option, 411
stochastic distributions, 477
seed; CIF simulator normal console output, 430
self loop, 66, 76
sensor
digital, 30
set, 109
difference, 110
intersection, 110
literal, 109
type, 109
union, 109
value, 109
simplify others, 708
preprocessing, 709
size considerations, 709
supported specifications, 708
simplify values, 709
implementation, 710
optimality, 711
preprocessing, 710
size considerations, 711
supported specifications, 709
simplify values (no references), 713
implementation, 714
optimality, 715
preprocessing, 713
size considerations, 715
supported specifications, 713
simplify values (no references, optimized), 715
implementation, 716
optimality, 716
preprocessing, 715
size considerations, 716
supported specifications, 715
simplify values (optimized), 712
implementation, 712
optimality, 713
preprocessing, 712
size considerations, 713
supported specifications, 712
simulation, 12, 399
model, 44
urgency, 422
simulation output, 427
console, 427
debug console output, 432
normal console output, 428
plot visualizer, 444
print output, 433
real-time, 451
state visualizer, 449
SVG visualization, 435
trajectory data, 441
simulator, 392
compiled code file, 400
Simulink; code generation, 549
single
state space
explorer, 728
generator, 728
specification, 56
ambiguity, 8
configuration, 84
contract, 12
formal, 11
incomplete, 9
inconsistent, 9
intuitive, 11
modular, 21
parameterization, 84
unambiguous, 11
state, 57, 397
intermediate, 452
invariant, 91
state/event exclusion invariant, 94
state space, 60, 398
state visualizer, 449
contents, 449
disable, 449
enable, 449
filtering, 450
icons, 450
legend, 450
multiple, 451
reset, 451
undo, 451
updates, 450
stochastics, 168
constant, 171
continuous, 170
discrete, 170
distribution, 168
sampling, 169
string, 102

literal, 102, 231
type, 102
value, 102
supervisor, 19, 19, 20, 22, 212
as plant, 696
empty, 37
implement, 36
monolithic, 39
synthesis, 19, 22, 34
validation, 35
verification, 35
supervisory controller synthesis, 212
SVG, 173
colors, 189
copy declaration, 251
declarations, 174, 248
id, 174
Inkscape, 176
input event, 415
input mapping, 264
input mode, 415
interaction, 198
interactive event, 415
interactivity, 415
move declaration, 255
output mapping, 258
text, 261
tspan, 261
tutorial, 174
viewer, 440
visualization, 173, 435
W3C, 173
XML, 174
SVG copy declaration, 251
application, 255
example, 251
file, 252
id, 252
order, 252, 255
overlap, 253
placement, 252
post, 252
postfix, 252
pre, 252
prefix, 252
root, 252
syntax, 252
uniqueness, 253
SVG declaration, 174, 248
push, 694
SVG copy declaration, 251
SVG input mapping, 264
SVG move declaration, 255
SVG output mapping, 258
SVG file declaration, 248
completeness, 250
path, 248
push, 694
scoping, 249
uniqueness, 250
SVG input mapping, 264
if event input mapping, 200, 265
single event input mapping, 199, 264
uniqueness, 266
update input mapping, 201, 265
SVG input mode, 415
debugging, 417
frame rate, 417
hovering, 415
interactive groups, 418
performance, 417
queue, 417
reset, 417
responsiveness, 417
simulation, 415
SVG move declaration, 255
application, 258
conflicts, 258
coordinates, 257
example, 256
file, 257
graphical elements, 258
id, 257
order, 258
syntax, 257
to, 257
x, 257
x coordinate, 257
y, 257
y coordinate, 257
SVG output mapping, 258
application, 263
attribute, 260
CSS style attributes, 260
escaping, 262
example, 258

file, 259
id, 259
order, 263
quoting, 262
spaces, 263
SVG presentation attributes, 260
syntax, 259
tabs, 263
text, 261
text labels, 261
type, 262
uniqueness, 263
value, 262
whitespace, 263
SVG viewer, 440
export, 441
GIF, 441
JPEG, 441
PNG, 441
refresh, 441
save as, 441
SVG, 441
SVG visualization, 435
application order, 436
Batik, 435
buffers/products example, 196
colors, 186
console output, 438
coordinate systems, 189
copy example, 192
copy of a copy, 196
debugging, 438
demo, 437
disable, 437
enable, 437
export, 439
first example, 175
grouping, 179, 180
lamps example, 192
literal values, 184
move example, 192
rate example, 184
record, 437
reset, 437
runtime error, 439
save as, 439
scalability, 186, 192, 196
scaling, 180, 189
simple solutions, 184
simulation, 435
stack trace, 439
sun/moon example, 179
SVG file declaration, 248
SVG standard, 173
tank example, 189
text formatting, 180
undo, 437
updating multiple attributes, 186
variable width, 186
visibility, 179
visualizer, 440
walk example, 180
workstation example, 186
y-axis invert trick, 189
switch, 114
expression, 114
switch expression
convert, 617
synchronization, 59
eliminate, 659, 669, 677
guards, 74
tau, 83
syntax
absolute name, 230
annotation name, 230
comments, 232
double, 231
grammar, 233
identifier, 229
keyword escaping, 229
keywords, 228
lexical, 228
name, 229
number, 231
relative name, 229
root name, 230
string, 231
terminals, 229
whitespace, 232
synthesis, 19
data-based, 306
example, 23
memory, 38
monolithic, 39
non-monolithic, 39
performance, 38

scalability, 39
supervisor, 22
tools, 34
system, 3
architecture, 4
controlled, 19, 22
cyber-physical, 3
layer, 4
multi-disciplinary, 9
sub-system, 4
uncontrolled, 17, 23

T

tau, 81
event, 81
explicit, 83
implicit, 83
synchronization, 83
tau event, 81
eliminate, 650
terminals, 229
test mode, 480
real-time simulation, 454
SVG debugging, 438
text
formatting, 208
text formatting, 208, 275
%%, 285
columns, 209
conversions, 276
escaping, 285
format patterns, 275
format specifiers, 276
introduction, 208
large numbers, 210
left alignment, 211
lower case, 276
quoting, 285
real values, 209
reference documentation, 275
unused values, 285
upper case, 276
time, 121
variable, 121
timed system, 49, 121
timing, 121
tools
CIF code generator, 514
CIF event disabler, 725
CIF explorer, 728
CIF information generator, 300
CIF merger, 716
CIF multi-level splitter, 360
CIF PLC code generator, 582
CIF relations collector, 302
CIF to CIF transformer, 600
CIF to mCRL2 transformer, 480
CIF to Supremica transformer, 386
CIF to UPPAAL transformer, 488
CIF to yEd transformer, 731
controller properties checker, 491
data-based supervisory controller synthesis, 306
event-based automaton abstraction, 384
event-based automaton projection, 382
event-based controllability check, 378
event-based DFA minimization, 381
event-based language equivalence check, 379
event-based NFA to DFA automaton conversion, 380
event-based nonconflicting check, 377
event-based observer check, 383
event-based supervisor synthesis, 371
event-based synchronous product, 370
event-based synthesis analysis, 373
event-based trim, 385
event-based trim check, 385
overview, 298
PLC code generator, 556
pretty printer, 603
scripting, 299
simulator, 392
text editor, 299
trace, 398
trace input mode, 413
trajectory data, 441
disable, 442
enable, 442
file, 442
filtering, 442
fixed output step size, 467
ODE solver, 443
output, 443
prettify, 443
real-time simulation, 467
reset, 444

time points, 443
types, 443
undo, 444
transition, 57
event, 122
time, 122
tuple, 103
eliminate field projections, 651
indexing, 104
literal, 103
packing, 104
projection, 104
type, 103
unpacking, 104
value, 103, 103
tutorial
language, 48
SVG, 174
TwinCAT, 572, 591
change mode, 573, 592
config mode, 572, 591
DUT, 573, 592
FAQ, 577, 596
generated output, 573, 592
GVL, 573, 592
PlcTask, 573, 592
POU, 573, 592
run mode, 572, 591
system modes, 572, 591
usage, 574, 593
type, 98
boolean, 102
declaration, 120
dictionary, 110
enumeration, 102
integer, 99
list, 105
real, 100
set, 109
string, 102
tuple, 103
type declaration, 120
eliminate, 652

U

ulp, 478
update, 69
urgency, 128
event, 128, 422
location, 128
simulation, 422

V

validation, 6, 8, 10, 12, 399
issues, 37
value, 98
boolean, 102
constant, 98
dictionary, 110
enumeration, 102
integer, 99
list, 105
literal, 98
real, 100
set, 109
string, 102
tuple, 103
variable, 127
?, 133
algebraic, 116
continuous, 122
different kinds, 127
discrete, 69
expression-local, 108
input, 165
new value, 78
old value, 78
read, 73
time, 121
versus location, 70, 71
write, 73
verification, 6, 8, 10, 12, 15
formal, 15
issues, 37
model checking, 15
void, 134
channel, 134

W

when, 69
whitespace, 232

Y

yEd
layout, 739
Structure View, 739