

MASTER

Industrial-strength formal verification of complex multi-disciplinary domain-specific models for embedded systems

Nelisse, J.J.

Award date:
2014

[Link to publication](#)

Disclaimer

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights

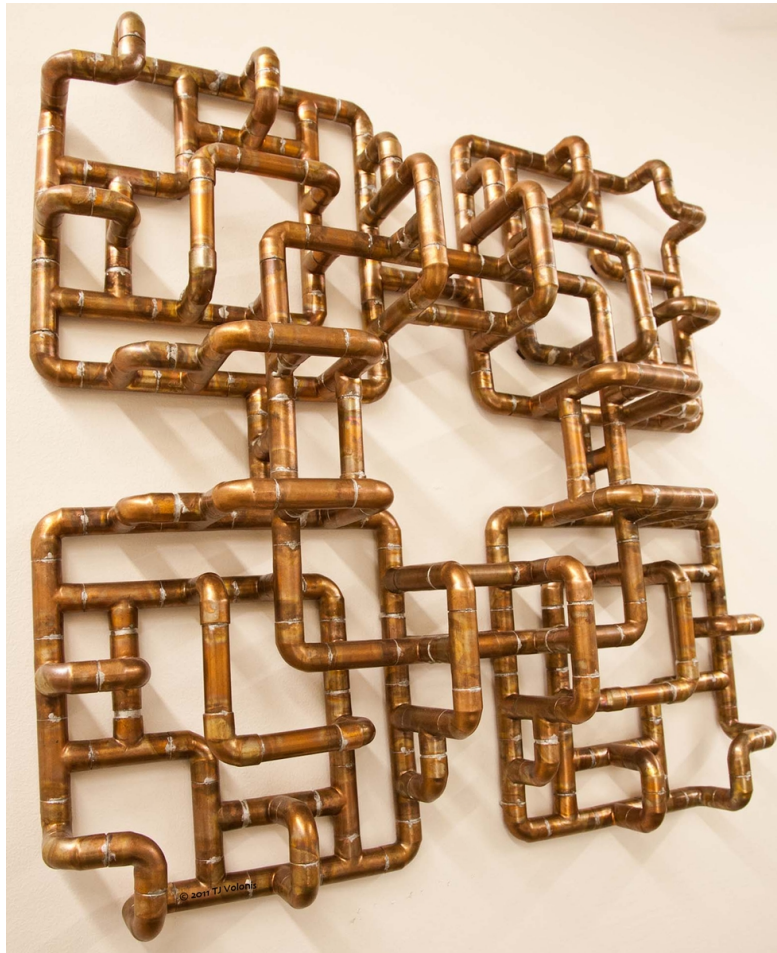
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

MASTER THESIS

Industrial-strength Formal Verification of Complex Multi-disciplinary Domain-specific Models for Embedded Systems

September 27, 2012



Author:

J.J. Nelisse

j.j.nelisse@student.tue.nl

Supervisor/Tutor (TU/e):

Prof. dr. ir. J.F. Groote

j.f.groote@tue.nl

External Supervisor (Sioux):

Ir. E. Schindler PDEng

eugen.schindler@sioux.eu

Abstract

In this thesis models are checked for correctness. The models describe a Home Heating System and a machine that is part of the solar panel production. Their components and configuration are modeled with a domain-specific Piping and Instrumentation diagram, while the behavior is modeled with restricted UML State Machines. For the model checking of the behavior a translation is made from these models to a formal verification system. This consists of an mCRL2 Specification and modal μ -calculus requirements. The translation is feasible, but the requirement specification and the model checking need quite some expert knowledge. Part of the results are guidelines for future modeling.

Contents

1	Introduction	6
1.1	Context	6
1.2	Problem	6
1.3	Cases	7
1.4	Approach	8
1.5	Summary of conclusions	8
2	Languages	9
2.1	Meta Model language	9
2.2	Piping and Instrumentation Language	10
2.2.1	Profile	10
2.2.2	Syntax	10
2.3	UML State Machine	13
2.3.1	Meta Model	13
2.3.2	Syntax	14
2.3.3	Semantics	15
2.4	mCRL2	16
2.4.1	mCRL2 Specification	17
2.4.2	Example	18
2.4.3	Linearization	19
2.5	Modal μ -calculus	20
2.5.1	Hennesy-Milner	20
2.5.2	Regular Formulas	20
2.5.3	Fixed Point Modalities	21
2.5.4	Modal Formulas with Data	21
2.5.5	Examples	21
3	Case descriptions	23
3.1	Language Workbench Challenge Case	23
3.1.1	Model	23
3.1.2	State Machines	24
3.2	SoLayTec Case	27
3.2.1	Model	28
4	Translation of Models to mCRL2	29
4.1	Translation Concept	29
4.2	Formal Translation	31
4.2.1	Language Workbench Challenge Design Decisions	32
4.2.2	SoLayTec Design Decisions	35
4.3	Parser	36
4.3.1	Trigger Parser	36
4.3.2	Constraint Parser	37
4.3.3	Effect Parser	39
4.4	Generator overview	40
5	Verification	41
5.1	Language Workbench Challenge Requirements	41
5.1.1	Liveness properties	41
5.1.2	Safety properties	49
5.2	SoLayTec Requirements	54
5.2.1	No deadlock	54
5.2.2	Reachability	55
5.2.3	No Interlocks	55
6	Guidelines for Modeling with Formal Verification in Mind	58
6.1	Guidelines for modeling	58
6.2	Guidelines for model checking	60
6.2.1	Properties	60
6.2.2	Tools	61

7	Related Work	63
8	Conclusions and Future Work	64
8.1	Problem Revisited	64
8.2	Results/Conclusions	64
8.3	Future Work	64
A	State Machines Meta Models	68
A.1	UML 2.4.1 State Machines	68
A.1.1	Behavior	68
A.1.2	NamedElement	68
A.1.3	Namespace	68
A.1.4	StateMachine	69
A.1.5	Region	69
A.1.6	Vertex	69
A.1.7	Pseudostate	69
A.1.8	PseudostateKind	69
A.1.9	ConnectionPointReference	69
A.1.10	State	69
A.1.11	FinalState	69
A.1.12	Transition	69
A.1.13	TransitionKind	69
A.1.14	Trigger	69
A.1.15	Constraint	69
A.2	Restricted UML State Machines	70
A.2.1	Model	70
A.2.2	Package	70
A.2.3	Class	70
A.2.4	Behavior	70
A.2.5	Element	70
A.2.6	NamedElement	70
A.2.7	StateMachine	70
A.2.8	Region	70
A.2.9	Vertex	70
A.2.10	Pseudostate	70
A.2.11	PseudostateKind	70
A.2.12	State	70
A.2.13	FinalState	71
A.2.14	Transition	71
A.2.15	Trigger	71
A.2.16	Constraint	71
B	Translation of Restricted UML State Machines to mCRL2	72
B.1	Notation	72
B.1.1	Meta Notation	72
B.1.2	Attributes	73
B.1.3	Functions	73
B.1.4	Global sets	73
B.2	Translation	73
B.2.1	Model	74
B.2.2	Package	74
B.2.3	ActiveClass	74
B.2.4	StateMachine	74
B.2.5	SortSpec	74
B.2.6	StateNames	74
B.2.7	StateName	75
B.2.8	InitialStateName	75
B.2.9	FinalStateName	75
B.2.10	Elements	75
B.2.11	Element	75
B.2.12	Instances	75

B.2.13	FunctionUpdates	75
B.2.14	FunctionUpdate	76
B.2.15	FunctionValues	76
B.2.16	SortSpecVarNames	76
B.2.17	ActSpec	76
B.2.18	ActionNames	76
B.2.19	ActionName	77
B.2.20	StateActionName	77
B.2.21	Trigger	77
B.2.22	Constraint	77
B.2.23	Behavior	78
B.2.24	ProcSpec	78
B.2.25	StateMachineParameters	78
B.2.26	StateMachineParameters2	78
B.2.27	StateMachineParameter	78
B.2.28	Region	78
B.2.29	Vertices	79
B.2.30	Vertex	79
B.2.31	Pseudostate	79
B.2.32	State	79
B.2.33	StateInitPre	80
B.2.34	StateInitPost	80
B.2.35	StateInit	80
B.2.36	FinalState	80
B.2.37	Transitions	81
B.2.38	Transition	81
B.2.39	Exits	82
B.2.40	NextStateMachineParameters	82
B.2.41	NextStateMachineParameters2	82
B.2.42	NextStateMachineParameter	82
B.2.43	NextCurrentStateName	82
B.2.44	Init	83
B.2.45	InitialStateMachineParameters	83
B.2.46	InitialStateMachineParameters2	83
B.2.47	InitialStateMachineParameter	83
B.2.48	InitialStateName	84
C	Toolsets	85
C.1	mCRL2 Toolset Overview	85
C.1.1	LPS Tools	85
C.1.2	LTS Tools	86
C.1.3	PBES Tools	86
C.2	LTSMIN Toolset	86

1 Introduction

1.1 Context

This thesis is the result of a Master project carried out as an internship at Sioux Embedded Systems B.V. in Eindhoven. This company provides solutions to high-tech companies in the area. Solutions are software, electronic, mathematical or part of secondment or consultancy. In more and more projects the paradigm of Model Driven Software Development (MDS) gains a foothold.

In MDS software models are used to design a system. The models are domain models, designed by a domain expert. For example, a software engineer is a domain expert in behavior and a liquid/gas flow expert is a Piping and Instrumentation domain expert. The domain experts have their own view on the system. They work together to create models, which describe the system. A domain language used in a model is defined by a meta model.

Often a software engineer is responsible for automatically generating code for a working system out of these models. When changes are made to the models, the new behavior is observed after regenerating the code. Figure 1 describes the current situation, where lessons about the behavior of the system are learned by inspecting the working system, which is run on a machine or in simulation.

Some benefits of this way of software development are that

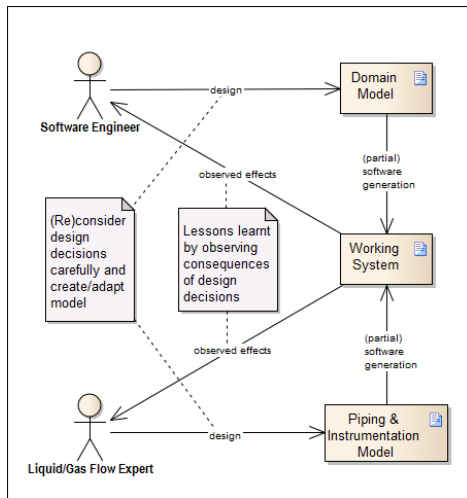


Figure 1: Domain experts interaction

- (almost) all the work is done on the model and not on code, because the model is closer to the domain and is used to generate code from. The models can be reused, hence, the development is more efficient and has higher quality.
- the domain expert models in a language that is close to the problem domain.
- the domain expert expresses exactly what he wants in the model. The domain expert is in control of the power and responsibility of development.
- by performing analysis on the model, instead of on the code, the quality of the model is higher, because the analysis is closer to the domain and thus the semantic level of analysis is higher.

1.2 Problem

The models used at Sioux are complete enough to make working systems of. However, models are more than just a nice abstraction for source code; there is a powerful potential in model checking. Namely that errors, such as safety violations, can be found before actual code is generated or used. Also insight in the behavior of a system can be obtained. Using the model checking outcome to correct the models, increases the quality of the system.

The problem investigated in this thesis is to see whether mCRL2, as a means for model checking, is useful in the context of Sioux to add model checking and analysis to model driven software development in a multi-disciplinary setting.

To make model checking possible an infrastructure needs to be made which has the following usability requirements:

- system requirements are specified by the domain expert, so each domain must have a suitable way to specify requirements and
- results of model checking must be understandable by the domain expert.

Furthermore the ingredients needed to make (meta) models suitable for formal verification have to be defined.

The proposed solution consists of the following steps:

1. translate the domain specific models and requirements to a formal verification system,
2. verify and analyze the formal model with respect to the formal requirements and
3. relate the results from the verification and analysis back to the domain-specific models.

Figure 2 shows this interaction of the domain expert with the formal verification system, where lessons are learnt from formal verification. The figure also shows that the effect of the changes made to the model, possibly due to the inconsistencies found in the system, should be observable after renewed verification and code generation.

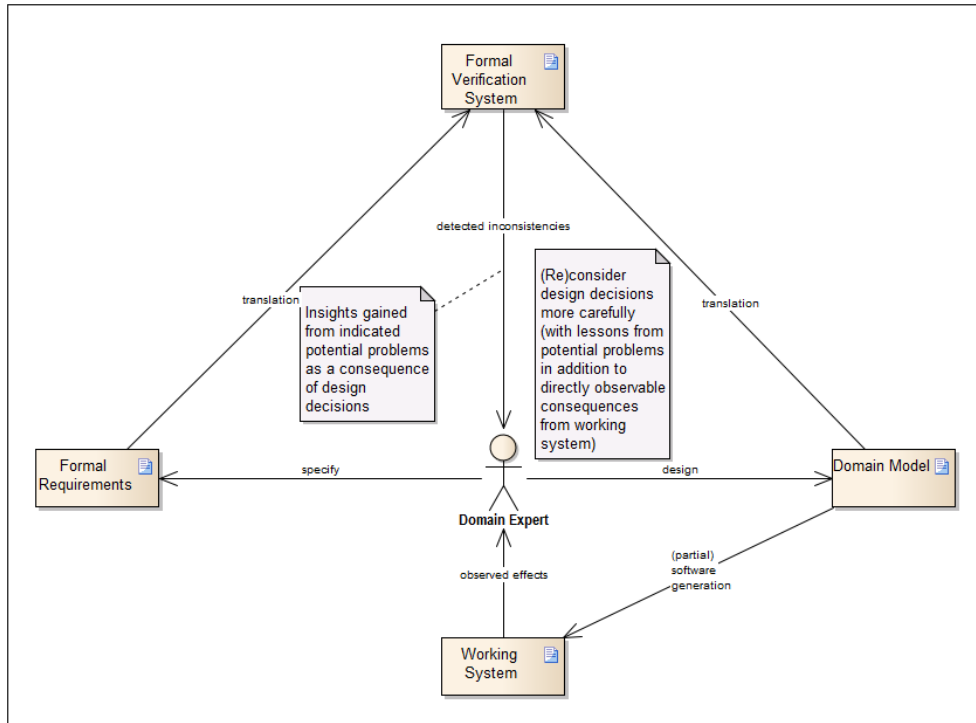


Figure 2: Domain expert interaction with formal verification

1.3 Cases

In this thesis, two cases are used to illustrate the work done. In the first case the models originate from the Language Workbench Challenge 2012 (LWC 2012) case [1] for the CodeGen 2012 Conference to which Sioux contributed. In the second case, the models are developed for SoLayTec, a company that makes machines which cover part of the process of solar panel production.

For the LWC 2012 case, the assignment was to apply MDSM in a non-software domain, controlled by software, hereby using / combining models based on multiple meta models. The non-software language Piping and Instrumentation (P&I) in general is used to design liquid and gas flow systems in industry, but in this case the language is used for the gas and liquid flow of a central heating system as can be found in houses. A domain expert in this area makes the models with the P&I language (an extension of the Unified Modeling Language (UML)). The syntax (notation) and semantics (meaning) are explained in Section 2.2.

Components that control the central heating system have behavior that is described with state machine diagrams (a part of UML). The syntax and semantics are explained in Section 2.3.

For the SoLayTec [28] case, also the P&I language and state machine diagrams are used to model the system. Due to intellectual property regulations, the (meta) models and requirements are not described in full detail.

1.4 Approach

Here the steps are listed, which are taken to come to the first parts of the infrastructure, namely the translation and the verification. During the steps concerning the translation a minimal example state machine, that contained the most important concepts, was used first, but later the Language Workbench Challenge 2012 (LWC 2012) and the SoLayTec models were used. The steps concerning verification, are applied to the LWC 2012 case, which is described in detail (Section 3.1) and SoLayTec case, a real life case, which is described very briefly (Section 3.2). The changes needed to make the models and meta models suitable for formal verification are recorded as guidelines for future modeling efforts (See Section 6).

1. A description of the syntax and the semantics of state machines is given (See Section 2.3).
2. A translation of the state machine syntax to the mCRL2 syntax is made on paper. This is done to get insight in the concepts of the translation and in the semantics of the state machine (See Section 4).
3. The translation is implemented and tested (See Section 4.1).
4. The translation is extended with the use of data, i.e., the (concrete) values of variables are seen as state of the modeled system. This way the behavior of the system is described at a more detailed level.
5. A few typical requirements to which the (behavior of the) system should adhere, are specified in modal μ -calculus (See Section 5.1).
6. These requirements are checked with the model using the mCRL2 toolset (See Section 5.1).
7. The results of the model checking process are related back to the domain-specific models, so the domain expert can understand them. This last step is important for usability of the proposed solution, but the automatic implementation is out of scope. However, it is tried to provide understandable explanations with each requirement and its model checking results.

1.5 Summary of conclusions

Translating the models used in the two case studies is feasible, but becomes significantly less straightforward when the formal model and the requirements (which both result from translation) must be formulated in such a way that the vast state spaces can be verified in an acceptable amount of time. Something that still needs further investigation is how to get the requirements specified in a domain-specific way, usable for the domain expert and how to give the model checking results back to the domain expert, also in a usable way.

2 Languages

In this section the modeling languages as used at Sioux, the formal verification language mCRL2 and the modal μ -calculus, a requirement specification language, are explained, since these languages will be used in the remainder of this thesis. First the meta model format is explained in Section 2.1. In this thesis two different meta models are used. One describes the Piping and Instrumentation Language (Section 2.2) and the other describes the State Machine Language (Section 2.3). Section 2.4 gives a brief description of the language mCRL2, which is the basic input for model checking and analysis. The formalism to denote requirements to be checked against the modeled system, called modal μ -calculus, is explained in Section 2.5.

2.1 Meta Model language

In Sections 2.2 and 2.3 some non-trivial textual notation is used to describe the syntax of state machines specified in the Uniform Modeling Language (UML) version 2.4.1 notation. The notation used and their relations are described in Table 2. For all other concrete syntax definitions of UML see [13].

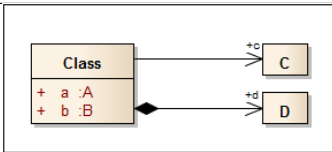
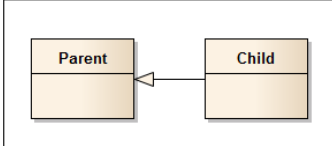
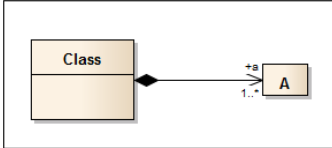
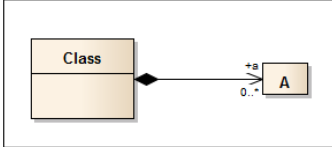
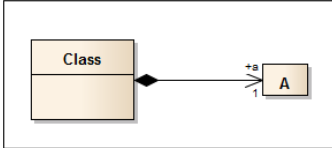
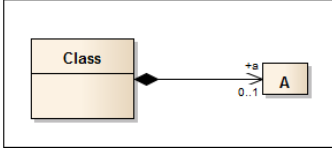
Semantics	Syntax (UML graphic)	Syntax (custom defined textual)
a class (with associations and attributes)		<i>Class</i> (<i>c</i> : <i>C</i> , <i>d</i> : <i>D</i> , <i>a</i> : <i>A</i> , <i>b</i> : <i>B</i>)
Child inherits from Parent (or Parent generalizes Child)		<i>Child</i> (<i>b</i> : <i>B</i> , inherits: { <i>Parent</i> })
a class with a collection of A attributes (multiplicity 1..* for A)		<i>Class</i> (<i>a</i> : <i>A</i> ⁺)
a class with an optional collection of A attributes (multiplicity 0..* for A)		<i>Class</i> (<i>a</i> : <i>A</i> [*])
a class with an A attribute (multiplicity 1 for A)		<i>Class</i> (<i>a</i> : <i>A</i>)
a class with an optional A attribute (multiplicity 0..1 for A)		<i>Class</i> (<i>a</i> : <i>A</i> [?])

Table 2: Meta model notation used in Section 2.2 and 2.3

A meta model is a model that describes the concepts of a (modeling) language and the relationships between them. The meta model records the syntax, i.e., how elements may or

must be connected and in what quantities.

Besides the use of a meta model to describe a language, a meta model can also be extended. For example a basic meta model that describes the basis concepts of a UML class diagram, is extended with domain specific concepts, to define a domain specific language. In this thesis this is done for the Piping & Instrumentation language. An extension to a meta model is called a *Profile*. With a Profile restrictions can be made, but also graphical notation can be added to the original meta model.

A profile denoted in UML can consist of the following elements:

metaclass a class whose instances are existing (meta model) classes

stereotype a class that extends a metaclass

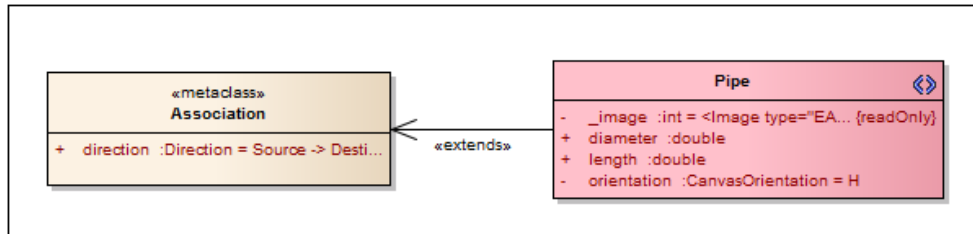


Figure 3: Profile example displaying metaclass and stereotype

In Figure 3 the metaclass ‘Association’ is extended by the stereotype ‘Pipe’.

The profile and (meta) models used in this document are made with Enterprise Architect 9.2 [29].

2.2 Piping and Instrumentation Language

In this section the Piping and Instrumentation (P&I) Language is described. Diagrams made with the Piping and Instrumentation Language are used in the process industry. The difference between a process in terms of mCRL2 and a process here, is that the first is a behavioral process, while the second is an industry process or procedure, concerning the processing of, for example, chemicals or raw materials. The diagram shows the piping of the process flow together with the installed equipment and instrumentation. It also shows the connections of the equipment and the instrumentation used to control the process. The diagrams play an important role in the specification, maintenance and modification of the described process.

In Section 2.2.1 the profile of the language is described and in Section 2.2.2 the concrete syntax elements of the language are shown and explained.

2.2.1 Profile

The profile shown in Figure 4 is an extension to a UML meta model. In this case it customizes a UML Class Model. Every stereotype also extends metaclass ‘Class’ directly. For readability, these arrows are not shown. Stereotype ‘Pipe’ extends metaclass ‘Association’ and stereotype ‘PILSModel’ extends metaclass ‘Package’. Dashed arrows indicate that the source stereotype uses the target enumeration. In the language there is an inheritance hierarchy, for example a ‘ThreeWayValve’ is also a ‘Valve’ and a ‘Valve’ is also a ‘Node’, which extends the metaclass ‘Class’. Note that some stereotypes are in italics. This means that they are abstract, i.e., cannot actually be applied to a class. ‘ThreeWayValve’, for example, can be applied to a class.

2.2.2 Syntax

The restricted Piping & Instrumentation elements used in this document and defined in the profile in Figure 4 are described here. There are also some elements that are not part of the standard that is defined for the language [18]. This profile describes a user made Domain Specific Language specifically tailored towards home heating systems. The visual representation of the described elements is shown in Table 3. Note that some elements did not get a domain specific representation, but are just represented by a normal class.

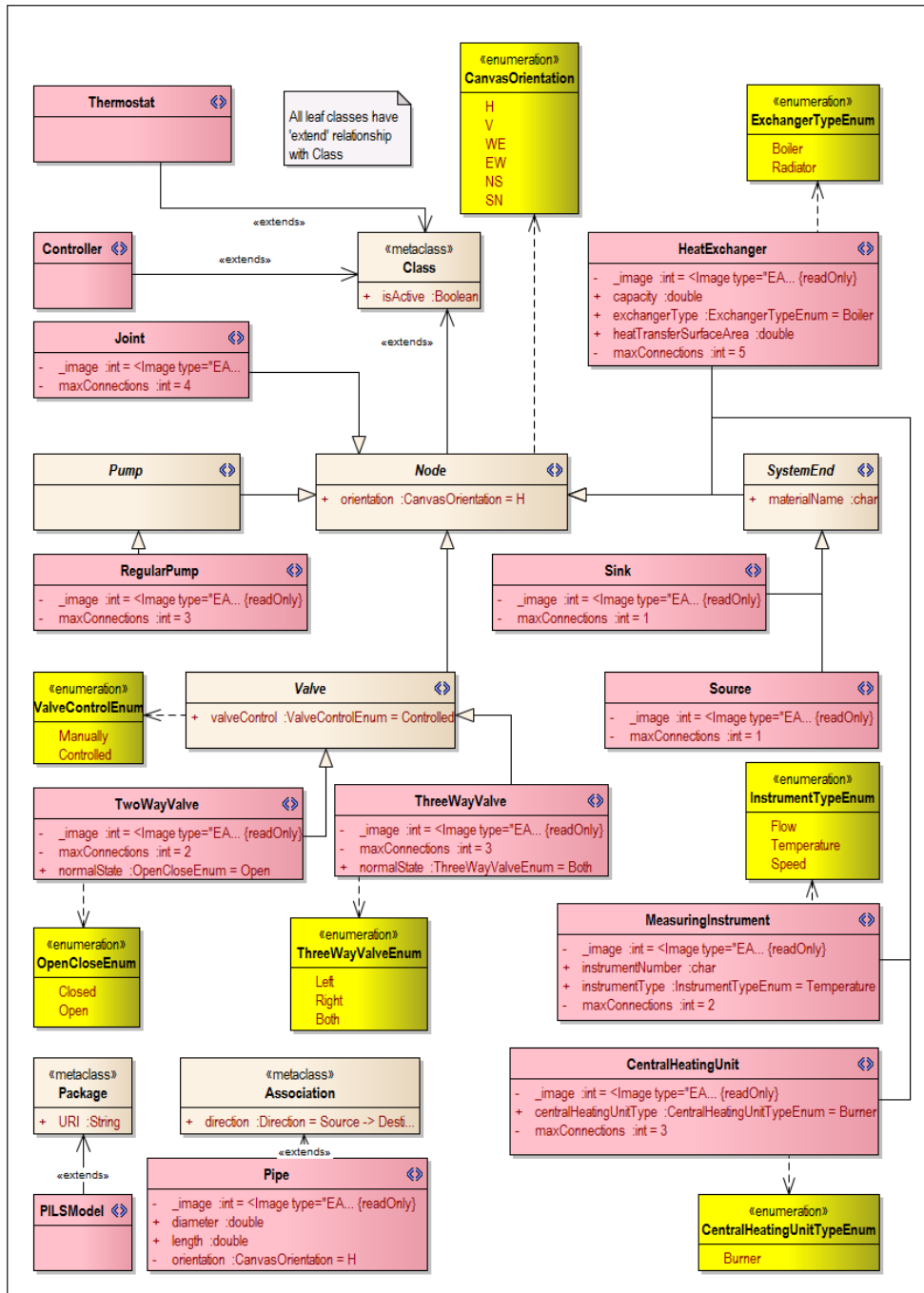
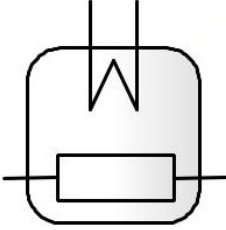
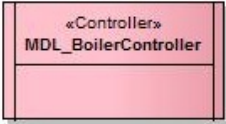
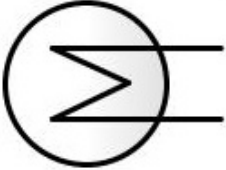


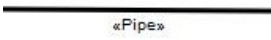





Figure 4: Piping and Instrumentation Profile

Element Name	Graphical Representation	Description
Central Heating Unit		A Central Heating Unit is a burner that increases the temperature of the water flowing through it. The Central Heating Unit is connected to three pipes, one for the water input, one for the water output and one for the gas input. The burner can be turned on or off.
Controller		A Controller controls the other elements in the diagram. This is done by a state machine as described in Section 2.3. The double vertical edge line denotes that the Controller is active.
Heat Exchanger		A Heat Exchanger is a boiler or a radiator exchanging the heat of the water to water or to the air respectively.
Joint		A joint connects two or more pipes with each other.
Measuring Instrument		A Measuring Instrument measures temperature, flow or speed of the resource in the element which it is connected to.
Pipe		A pipe connects other elements and is used to let gas or water flow through it.
Regular Pump		A Regular Pump transports the gas or water in a certain direction. It is connected to three pipes, one for the input, one for the output and one for the flow Measurement Instrument. It can be switched on and off.
Sink		A Sink is an output of the system, for example connected to a valve to tap off water. It is connected to one pipe.
Source		A Source is an input of the system and provides external resources to the system. It is connected to one pipe.

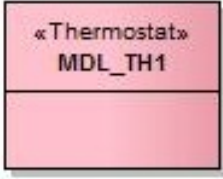
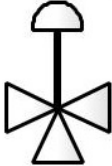


Element Name	Graphical Representation	Description
Thermostat		A Thermostat contains a temperature sensor and a set point unit. The behavior connected to this is managed by a separate controller.
Three Way Valve		A Three Way Valve connects three pipes. One is input and two are output. The valve can be opened to the left, the right or to both sides. So it is always open.
Two Way Valve (manual)		A manual Two Way Valve connects two pipes. The valve can be opened or closed.
Two Way Valve (controlled)		A controlled Two Way Valve connects two pipes. The valve can be opened or closed by a Controller.

Table 3: Piping and Instrumentation Language elements

2.3 UML State Machine

In this Section the UML state machine language that is used in this document is explained. Section 2.3.1 describes the meta model that defines the abstract syntax of the state machine language. In Section 2.3.2 the concrete syntax, i.e., the graphical representation of state machines, is found. The semantics are described in Section 2.3.3. More about the UML state machine language is found in Appendix A and [23].

2.3.1 Meta Model

In this section the UML meta model of the state machine language is described. It is based on the UML version 2.4.1 standard [13] defined by the Object Modeling Group (OMG) [23]. In this document the state machine language is not used in its full extent. Some adaptations have been made, with respect to the standard. They make model checking easier. These adaptations are:

- An initial state may have multiple outgoing transitions.
- A state machine or composite state may have only one region.
- A pseudostate can only be an initial state.
- Sub state machines, i.e., references to other state machines, are not allowed, and thus also entry and exit points of a state machine are not allowed.

The description of the notation used in Figure 5 is found in Section 2.1. In Figure 5 are, besides the state machine meta model, the classes ‘Model’, ‘Package’ and ‘Class’ added, to

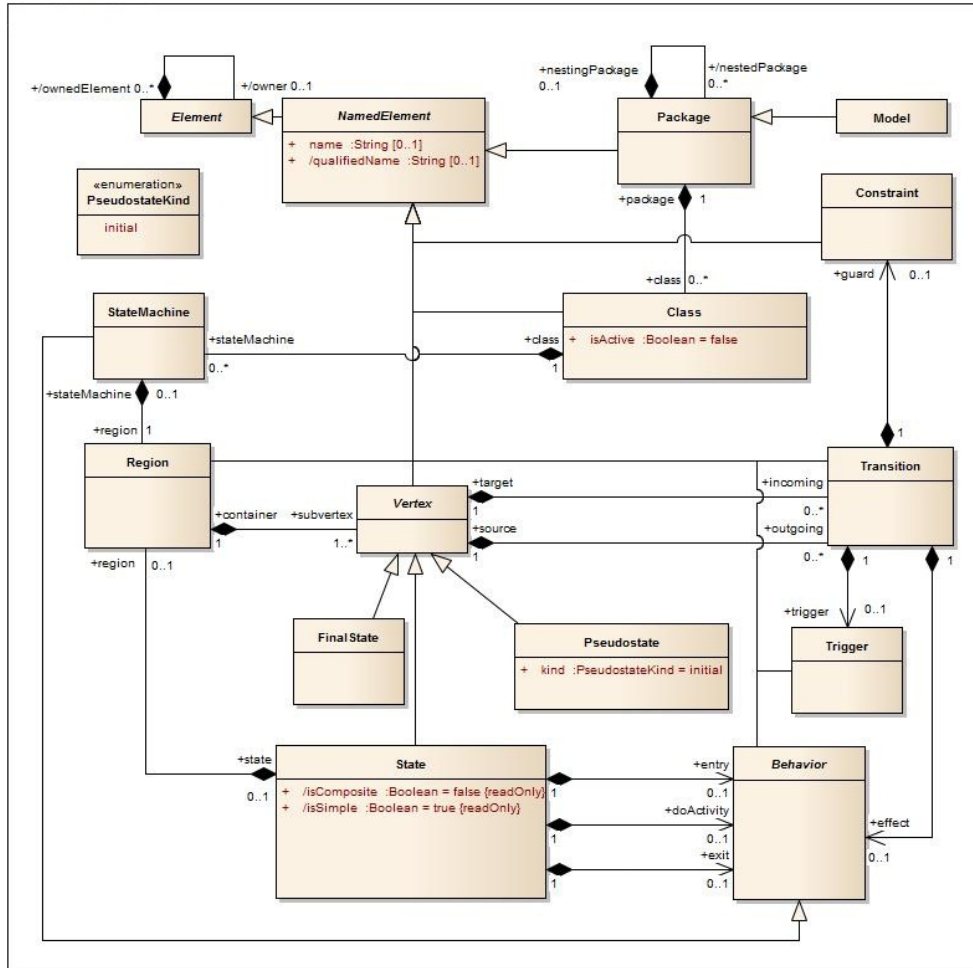


Figure 5: Simplified UML State Machine Meta Model

represent the hierarchy as used in the modeling program Enterprise Architect. So a ‘Model’ is a special ‘Package’, a ‘Package’ contains nested packages or classes and a ‘Class’ can contain any number of state machines. The inheritance and UML package relations that are not used in this document are left out.

2.3.2 Syntax

The concrete syntax elements of the state machine language is explained by means of the state machine diagram in Figure 6. The numbers (i) in the following element descriptions correspond to the numbers in the figure:

- A *state machine*, which is denoted by the rectangular bounding box (1).
- A state machine contains one *region*, which is the workspace of the state machine, but has no graphical representation.
A region can contain *states*, *transitions*, an *initial state* and possibly a *final state*.
- A *state* denotes a state of the state machine. A state can be either *composite* (2), meaning that it contains a region, such as described above, or *simple* (3), meaning that it contains no region. States can be nested, using a composite state, since its region can contain states.
A state can also have behavior in it, *entry*, *do* and *exit* behavior.
- *Entry behavior* (4) is performed when the state is entered.
- *Do behavior* (5) is performed when the state is active.
- *Exit behavior* (6) is performed when the state is exited.

Source	Target	LCA	Hierarchy
Initial	STATE1	StateMachine	StateMachine
STATE1	STATE1	StateMachine	Initial
STATE1	STATE2	StateMachine	STATE1
STATE2	STATE3	StateMachine	STATE2
Initial	STATE2_1	STATE2	Initial
STATE2_1	STATE2	StateMachine	STATE2_1
STATE2_1	STATE2_2	STATE2	STATE2_2
STATE2_2	STATE2_1	STATE2	STATE3
STATE2_2	STATE3	StateMachine	Initial
STATE2_2	STATE3_2	StateMachine	STATE3_1
STATE3	STATE4	StateMachine	STATE3_2
Initial	STATE3_1	STATE3	STATE4
STATE3_1	STATE3_2	STATE3	
STATE3_1	Final	StateMachine	
STATE3_2	STATE3_1	STATE3	
STATE3_2	STATE2_2	StateMachine	
STATE4	STATE1	StateMachine	

Table 4: Least Common Ancestors (left) and hierarchy (right) of the states of the state machine in Figure 6

3. In a state perform the entry behavior if in the previous round a transition was performed.
4. In a state perform the do behavior.
5. If this round no transition is performed yet, check the transitions in order of appearance in the state machine.
6. If a transition is enabled, i.e., the trigger is set and the guard is true, update the state and perform the effect behavior.
7. Continue with (2) if state is composite, i.e., contains a region.
8. If during this round a transition was performed perform exit behavior.

Since the behavior is recursively defined, the entry behavior and do behavior is performed from the outside to the inside, and the exit behavior from the inside to the outside. For transitions priority is dependent on the LCA of the source and target state of the enabled transitions. The LCA higher in the hierarchy has higher priority than the LCA lower in the hierarchy. If no transitions are enabled, the state machine stays in the same state, possibly only executing its entry and do behavior.

If more than one transition is enabled on the same level, i.e., they have the same LCA, the transition that is checked first is taken, the order is determined by the order in which the transitions are defined in the state machine diagram. Thus if the diagram is constructed in a different order the resulting code and thus its behavior is most likely not the same. Note that all behavior, i.e., entry, do, exit and effect behavior, is only performed if it is present. Also the checks, i.e., trigger and guard, for a transition are only done if they are present.

2.4 mCRL2

In this section the language mCRL2 is explained briefly, because this is the formal language to which the models are translated, to make formal verification possible. The abbreviation mCRL2 stands for micro Common Representation Language 2. The language is used for the specification, analysis, verification and validation of the behavior of communicating processes and protocols. The language is based on the Algebra of Communicating Processes (ACP) [3], extended with data and time. An overview of the toolset that accompanies the language can be found in Appendix C.1.

2.4.1 mCRL2 Specification

A specification in the mCRL2 language consists of data, actions, processes and an initial process. The data specification consists of sorts, constructors, functions, variables and data equations.

Sorts are data types that can be used throughout the whole specification. There are some basic sorts built into the language, such as `Bool`, `Real` and `Int`. A sort is defined with the keyword `sort` and constructors for a sort with the keyword `cons`. For example the sort `Bool` is defined as follows:

```
1 sort Bool;
2 cons true, false: Bool;
```

Constructors are not the only way to define a sort, a structured sort is defined by using the `struct` keyword and a function sort, denoting the sort of all functions over the used sorts, uses an arrow:

```
1 sort States = struct Initial | S_0 | S_1 | Final;
2 sort Int2Bool = Int -> Bool;
```

Sorts often have operators and functions defined on them, some are built in, such as comparison operators on sorts and others have to be defined. An example is a function for the ‘exclusive or’ on two booleans:

```
1 map xor: Bool # Bool -> Bool;
2 var b1, b2: Bool;
3 eqn xor(b1,b2) = if(b1, if(b2, false, true), b2);
```

The keyword `map` declares the function (or mapping). The keyword `var` defines some variables that are used in the function definition, which is denoted with the keyword `eqn`. The ‘if’ function takes a boolean expression as first argument and if the expression is true it returns the second argument and if it is false it returns the third argument.

Actions are declared with the keyword `act`. Actions can contain data parameters. There is also an internal action ‘tau’, which is used to denote actions not observable for the outside world. Two or more actions can be combined to a multi action with a bar (‘|’).

A Process specification starts with the keyword `proc` followed by its name and possible data parameters. A process can be made up of the following constructs:

- *(Multi)actions*
- The process that does nothing, i.e., it deadlocks or terminates unsuccessfully. This is denoted by *delta*.
- The *sequential composition* of processes is the sequential execution of two processes. The process term $a . b$ denotes that action a happens first and after that action b happens.
- The alternative composition of processes is the nondeterministic choice of two processes. The process term $a + b$ denotes that either action a happens or action b happens.
- A *summand* over processes (generalized alternative composition) is the general form of the nondeterministic choice of processes. The choice is determined by the data parameter(s) over which the summand ranges. The process term $\text{sum } b:\text{Bool} . c(b)$ is a shorthand for $c(\text{true}) + c(\text{false})$.
- A *conditional choice* of processes is used to guard processes. The process term $(b) \rightarrow c \langle \rangle d$, denotes that if b is true, the process continues with action c and otherwise with action d .
- The *parallel composition* of processes denotes that the actions of the parallelized processes are interleaved, i.e., occur after each other, and communicate, i.e., happen simultaneously. This communication is denoted by a multi action when the parallel processes are unfolded. For example $a . b \parallel c . d$ denote the parallel composition of process $a . b$ with process $c . d$, unfolded it is written as $a.(b.c.d+c.(b.d+b|d)+b|c.d)+c.(a.(b.d+b|d)+d.a.b+a|d.b)+a.c.(b.d+d.b+b|d)$ representing all combinations possible when interleaving and synchronizing the actions. Here, for example, the synchronized actions b and d are denoted by the multi action $b|d$.

- *Recursion* is used to execute the same process again, with possibly altered parameters. For example **proc** B = a.B + c, denotes that after doing an a action the same process is executed again.

As seen above, the unfolding of the parallel composition of two simple process, leads easily to a quite complicated process. To cope with that complexity there are operators that can help to reduce the process. These are:

- the communication operator, which takes synchronized (communicating) actions and gives them a single action name. A process term looks like **comm**({a|c -> e}, a . b || c . d). Note that communication only takes place if also the data parameters are the same.
- the blocking operator, which blocks actions. Blocking means that the actions are rewritten to delta. The blocking operator is denoted by **block**(B, ...), where B is the set of actions that are blocked.
- the allowing operator, which allows actions (and disallows all others). A process term using this is **allow**({e, b, d}, ...). Note that a multi action to be allowed, should be provided explicitly.
- the hiding operator, which hides actions from the outside world, i.e., makes the actions internal. This is denoted by **hide**({b}, ...).
- the renaming operator, which renames an action to another action name. The number and sorts of the parameters of the defined actions, must be equal. For example **rename**({a -> b}, ...) renames the action a to action b.

Note that the parallel operators may only be used in the initial process. Such initial process is executed by using the **init** keyword, as in **init** B(*true*); stating that process B will start with its parameter set to *true*.

The language contains also the possibility of modeling timed behavior. For this and more detailed information see [7, 12].

2.4.2 Example

To illustrate the described language constructs consider the following example of an mCRL2 Specification:

```

1 sort State = struct Initial | State1 | State2 | Final;
2 act Initial, Final;
3   Transition1: Bool;
4   Transition2;
5   read, send, exchange: Bool;
6 map isGreaterThanZero: Int -> Bool;
7 var i: Int;
8 eqn isGreaterThanZero(i) = i > 0;
9 proc Process(currentState: State) =
10   (currentState == Initial) -> Initial . Process(currentState = State1)
11 + (currentState == State1) -> sum j: Int . (-10 < j && j < 10) ->
12   Transition1(isGreaterThanZero(j)) .
13   Process(State2)
14 + (currentState == State2) -> (sum boolExpr: Bool . (boolExpr) -> read(boolExpr) .
15   Transition2 . Process(State2) <> read(boolExpr) .
16   Process(Final))
17 + (currentState == Final) -> Final . delta;
18 proc Var(i: Int) = send(i < 2) . Var(i+1);
19 init block({send, read}, comm({send|read -> exchange}, Process(Initial) || Var(0)));

```

In Line 1 of the example above a structured sort is defined which represents state names used in the process Process.

In Lines 2 till 5 the actions ‘Initial’, ‘Final’ and ‘Transition2’ are defined without data parameter and the actions ‘Transition1’, ‘read’, ‘send’ and ‘exchange’ are defined with one boolean data parameter.

The function ‘isGreaterThanZero’, taking an integer value and returning a boolean value is declared and defined in Lines 6 till 8.

The process ‘Process’, defined in Lines 9 till 17, has one data parameter representing the current state of the process.

When the process is in state ‘Initial’ it can execute action ‘Initial’ and then does a recursive call to ‘Process’, whereby the current state is updated to state ‘State1’.

In state ‘State1’ a summand over the integer parameter ‘j’ is used to give a value to the data parameter of ‘Transition1’, namely ‘isGreaterThanZero’ for integer parameter ‘j’. Since the integer numbers are infinite, the value of ‘j’ is bounded by -10 and 10. This could also be a smaller range, for example 0 and 1 as bounds would also do. The value of ‘isGreaterThanZero(j)’ has two possible values, which can both be reached with the integers and thus the summand only yields two different transitions, which can occur multiple times: ‘Transition1(true)’ and ‘Transition1(false)’. After such transition the process ‘Process’ continues with state ‘State2’.

In state ‘State2’ a summand over boolean parameter ‘boolExpr’ is used to guard the subsequent processes. If ‘boolExpr’ is *true* then a ‘read’ action with data parameter ‘boolExpr’, i.e., *true* is executed after which ‘Transition2’ is executed and the process ‘Process’ continues in state ‘State2’. If ‘boolExpr’ is *false* then a ‘read’ action with data parameter ‘boolExpr’, i.e., *false* is executed and it will continue in state ‘Final’.

In state ‘Final’ the action ‘Final’ is performed and then the process continues with the ‘delta’ process, meaning that it does nothing anymore.

Process ‘Var’ with one integer parameter, denoting a counter, executes action ‘send’ with data value *true* if ‘i > 2’ and *false* if ‘i ≤ 2’. The process continues with parameter ‘i’ increased by one.

The initial process in Line 18 first puts processes ‘Process’ and ‘Var’, with their data parameter initially set to ‘Initial’ and ‘0’ respectively, in parallel. The multi action ‘send|read’ must communicate to action ‘exchange’, all other occurrences of ‘send’ and ‘read’ actions are blocked, leaving only actions ‘Initial’, ‘Transition1’, ‘Transition2’, ‘exchange’ and ‘Final’ available.

2.4.3 Linearization

An mCRL2 Specification needs to be linearized to be used by most of the tools in the mCRL2 toolset. This linearization yields a Linear Process Specification (LPS). An LPS is a restricted form of an mCRL2 Specification. With such an LPS a Labeled Transition System (LTS) can be generated. This can best be done if the data used in the specification is finite, because if it is infinite, it takes a long time.

An LTS is a directed graph, where the vertices represent states, and the edges represent transitions. A transition contains a label. The action (names) in an mCRL2 Specification correspond to these labels. The values of the parameters in the processes and the position in the process while executing define the state, so in the example of Section 2.4.2 the so called state vector is made up of the values of ‘currentState’, ‘i’ and the positions of processes ‘Process’ and ‘Var’. The LTS of the example above is shown in Figure 7.

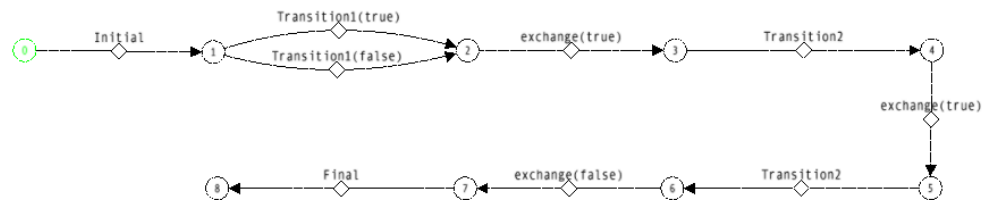


Figure 7: Labeled Transition System corresponding to the example specification of Section 2.4.2

Verification with the mCRL2 toolset is mainly done with the LPS of an mCRL2 Specification. In Section 2.5 the formalism is explained that is used to specify requirements concerning states, but also concerning transitions and paths.

2.5 Modal μ -calculus

In this Section the modal μ -calculus is described, in Sections 2.5.1 till 2.5.4 the language is built up. In Section 2.5.5 some examples of properties stated in the μ -calculus are given and explained.

2.5.1 Hennesy-Milner

The modal μ -calculus is a requirement specification language. The language is based on Hennesy-Milner logic [17]. The *Hennesy-Milner logic* has the following syntax grammar:

$$\phi ::= \text{true} \mid \text{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \langle a \rangle \phi \mid [a]\phi$$

These modal formulas say something about a state. The modal formula *true* is true in each state of a process (see Section 2.4.3) and *false* is never true in a state. The connectives \wedge (and), \vee (or) and \neg (not) have their usual logical meaning. It is also allowed to use other propositional logic connectives as \Rightarrow (implication) and \Leftrightarrow (bi-implication), because they can be expressed in the connectives described above. The diamond modality $\langle a \rangle \phi$ holds whenever an a -action can be performed, and ϕ holds afterwards. The box modality $[a]\phi$ holds in a state, when after each a -action that can be performed from that state, afterwards ϕ holds. The $\langle a \rangle \phi$ and $[a]\phi$ formulas clearly express different properties. They can be combined with other constructs to formulate more complex properties or requirements.

2.5.2 Regular Formulas

To be able to express properties over more than one action, regular formulas can be used within modalities. They are based on action formulas, having the following syntax:

$$\alpha ::= \alpha_1 \mid \dots \mid \alpha_n \mid \text{true} \mid \text{false} \mid \bar{\alpha} \mid \alpha \cap \alpha \mid \alpha \cup \alpha.$$

An action formula defines a set of actions. So the formula $\alpha_1 \mid \dots \mid \alpha_n$ defines the set with only that multi action in it. The formula *true* denotes the set of all actions in the process and *false* then represents the empty set. The connectives \cap, \cup denote intersection and union of sets of action. The complement of the set of actions α is denoted by $\bar{\alpha}$. This is done with respect to the set of all actions.

The definitions of modalities with action formulas is the following.

$$\langle \alpha \rangle \phi = \bigvee_{a \in \alpha} \langle a \rangle \phi \quad [\alpha] \phi = \bigwedge_{a \in \alpha} [a]\phi,$$

where α is a set of actions.

Regular formulas extend the action formulas to allow the use of sequences of actions in modalities. The syntax of regular formulas is given by the following grammar.

$$R ::= \epsilon \mid \alpha \mid R \cdot R \mid R + R \mid R^* \mid R^+.$$

The formula ϵ represents the empty sequence of actions. For this it holds that $[\epsilon]\phi = \langle \epsilon \rangle \phi = \phi$, meaning that it is always possible to perform no action and thus staying in the same state. The formula α denotes an action formula. The regular formula $R_1 \cdot R_2$ denotes the concatenation of the sequences represented by R_1 and R_2 . The formula $R_1 + R_2$ expresses the union of the sequences in R_1 and R_2 . Their definition is as follows:

$$\begin{aligned} \langle R_1 + R_2 \rangle \phi &= \langle R_1 \rangle \phi \vee \langle R_2 \rangle \phi & [R_1 + R_2] \phi &= [R_1] \phi \wedge [R_2] \phi \\ \langle R_1 \cdot R_2 \rangle \phi &= \langle R_1 \rangle \langle R_2 \rangle \phi & [R_1 \cdot R_2] \phi &= [R_1][R_2] \phi. \end{aligned}$$

The regular formulas R^* and R^+ allow for iterative behavior and denote zero or more repetitions and one or more repetitions of the sequences in R respectively.

Two formulas are commonly used: the always and eventually modalities. The always modality is often denoted as \square and expresses that ϕ holds in all reachable states. The eventually modality is denoted as \diamond and expresses that there is a sequence of actions that leads to a state in which ϕ holds. They can be expressed in regular formulas as follows:

$$\square \phi = [\text{true}^*] \phi \quad \diamond \phi = \langle \text{true}^* \rangle \phi.$$

The always modality is a typical instance of a safety property, saying that something bad will never happen. The eventually modality is used in liveness properties, saying that something good will eventually happen.

2.5.3 Fixed Point Modalities

Regular expressions are very expressive and suitable to state most behavioral properties, but cannot serve every purpose. The Hennessy-Milner logic can be extended by adding explicit minimal and maximal fixed point operators. This is called the modal μ -calculus. Its expressiveness is shown by the fact that all regular formulas can be translated in the modal μ -calculus, but the expressiveness also makes it far from easy to formulate properties using the modal μ -calculus. The syntax of the basic form of the modal μ -calculus is given by the following grammar:

$$\phi ::= \text{true} \mid \text{false} \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \langle a \rangle \phi \mid [a] \phi \mid \mu X. \phi \mid \nu X. \phi \mid X.$$

The formula $\mu X. \phi$ is the minimal fixed point and $\nu X. \phi$ denotes the maximal fixed point. Variable X ranges over strings. To understand fixed point modalities, X can be considered as a set of states. The formula $\mu X. \phi$ holds in all those states in the smallest set X that satisfies the equation $X = \phi$, where it is likely that X occurs in ϕ . Stated otherwise, think of X as the set of states where ϕ holds. In the same way, $\nu X. \phi$ holds for the states in the largest set X that satisfies $X = \phi$.

Since safety properties, stating that something bad will never happen, have to hold in the whole system, the maximal fixed point operator can be used, since it ‘looks’ for the largest set satisfying the property. Conversely the minimal fixed point can be used for liveness properties, stating that something good will eventually happen, since it is already satisfied if it holds for a smallest set.

By nesting fixed point operators, so-called *fairness* properties can be expressed, saying that some action must happen, provided that it is unboundedly often enabled or because some other action happens only a bounded number of times.

2.5.4 Modal Formulas with Data

Processes in mCRL2 are extended with data and in a similar way modal formulas are extended with data, and sometimes time, to describe the real world more closely. The extension is done in three ways:

1. modal variables can have arguments,
2. actions can carry data arguments and time stamps and
3. existential and universal quantification is possible.

This results in the following syntax, where α denotes a multi-action, af stands for an action formula, R denotes a regular formula and ϕ represents a modal formula.

$$\begin{aligned} \alpha &::= \tau \mid a(t_1, \dots, t_n) \mid \alpha \mid \alpha. \\ af &::= t \mid \text{true} \mid \text{false} \mid \alpha \mid af \mid af \cap af \mid af \cup af \mid \forall d : D. af \mid \exists d : D. af. \\ R &::= \epsilon \mid af \mid R \cdot R \mid R + R \mid R^* \mid R^+. \\ \phi &::= \text{true} \mid \text{false} \mid t \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \forall d : D. \phi \mid \exists d : D. \phi \mid \langle R \rangle \phi \mid [R] \phi \mid \\ &\quad \mu X(d_1 : D_1 := t_1, \dots, d_n : D_n := t_n). \phi \mid \\ &\quad \nu X(d_1 : D_1 := t_1, \dots, d_n : D_n := t_n). \phi \mid X(t_1, \dots, t_n). \end{aligned}$$

Any expression t of sort \mathbb{B} is an action formula. if t is true, it represents the set of all actions and if it is false it represents the empty set. The action formula $\exists d : D. af$ represents $\bigcup_{d:D} af$. Dually, $\bigcap_{d:D} af$ is a representation for $\forall d : D. af$. Also the normal meaning of the universal and existential quantification over data can be used in the modal formulas, i.e., $\forall d : D. \phi$ is true, if ϕ holds for all values from the domain D substituted for d in ϕ . For the existential quantifier, it works similar.

For more and detailed information see [12].

2.5.5 Examples

A typical example of a safety property is the statement that a deadlock should be absent. In the modal μ -calculus this is written as:

$$[\text{true}^*] \langle \text{true} \rangle \text{true}$$

Here $[true^*]$ means ‘in every reachable state’ and $\langle true \rangle true$ means ‘there exists a transition after which $true$ holds’. Because $true$ holds in every state, this can also be stated as just ‘there exists a transition’. Concluding the statement says: ‘In every reachable state there exists a transition’. If that is indeed the case, then there is of course no deadlock, because that would mean that there is a state for which there exists no transition and thus contradicting the statement.

In systems the situation that something happens and after that something else should happen is called a liveness property. An example is:

$$[true^* \cdot send] \mu X. [\overline{read}] X \wedge \langle true \rangle true$$

Here $[true^* \cdot send]$ means ‘in every reachable $send$ state’ and $\mu X. [\overline{read}] X \wedge \langle true \rangle true$ means ‘the read action must be done anyhow’. Note that the $\langle true \rangle true$ part enforces that the formula is not also true if a deadlock occurs. Concluding it states that after a send action a read action must be done. This formula is not true if there is a loop in the system without a read action, which can be done infinitely often and thus never reach the read action.

A fairness property says that an action must happen provided that this action is unboundedly enabled, or because another action happens only a bounded number of times. Consider the formula:

$$\mu X. \nu Y. ((\langle read \rangle true \wedge [send] X) \vee (\neg \langle read \rangle true \wedge [send] Y))$$

This says that on an infinite send path, only a finite number of states have read actions enabled. This is caused by the minimal fixed point before the X , which allows for finite traversal, which in this case only happens when a read action is enabled. The maximal fixed point before the Y , allows to traverse the $send$ action infinitely often, whereby a read action is never enabled.

Also data can be used in formulas, for example in this formula:

$$[true^* \cdot \exists n : \mathbb{N}. error(n)] \mu X. ([\overline{shutdown}] X \wedge \langle true \rangle true)$$

which states that a if an error with some error number n occurs, a shutdown is inevitable.

3 Case descriptions

In this section the two cases as used in this thesis are explained. In Section 3.1 the Language Workbench Challenge case is described, and Section 3.2 contains the description of the SoLayTec case.

3.1 Language Workbench Challenge Case

The Language Workbench Challenge 2012 Case [1] described here, is part of the Sioux contribution to the CodeGen 2012 Conference. The assignment consisted of a description of a Home Heating System, for which a meta model and model had to be defined. Furthermore state machines had to be developed to control the system. Enterprise Architect was used to build the meta model (profile), model and state machines. A Programmable Logic Controller (PLC) Simulator was used to execute generated (and handmade) code to simulate a working Home Heating System. In such a PLC every few milliseconds a run is made over all the PLC code. So for each state machine the steps belonging to one state are executed. In Section 3.1.1 the model as specified in the case is given and explained and in Section 3.1.2 the state machines are described.

3.1.1 Model

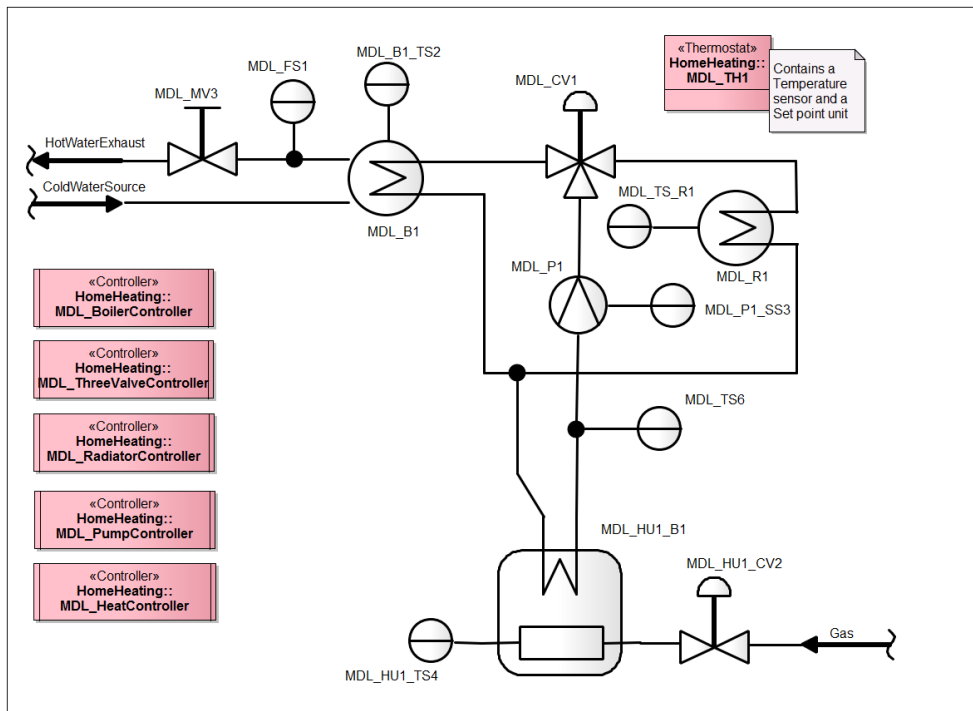


Figure 8: Home Heating Model

The model that describes the Home Heating System in the Piping and Instrumentation Language can be found in Figure 8. The system has inflow of gas and water and outflow of water. There are a manual valve, a three way valve and a controlled valve, a pump in the center and two heat exchangers, namely a boiler and a radiator. Furthermore there is one central heating unit at the bottom and a thermostat. Several elements have a measuring instrument, measuring the flow, temperature or the speed, connected to it. All elements are connected by pipes and joints.

The model contains several controllers. Every controller has a state machine connected to it, describing the behavior of the controlled element. The state machines are shown in Figure 9 till 12.

3.1.2 State Machines

This section describes the state machines connected to the controller elements of the model.

BoilerController

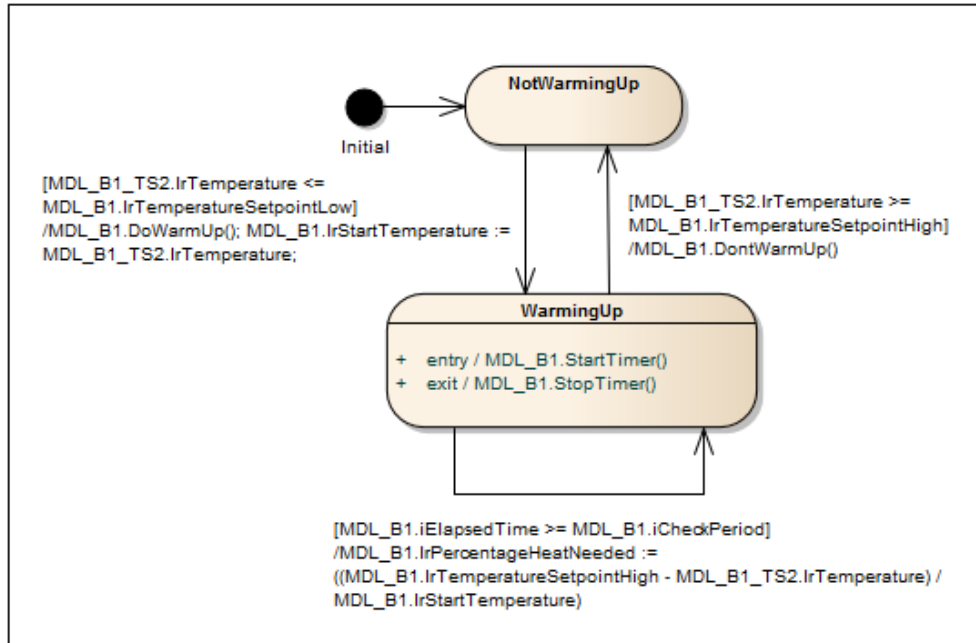


Figure 9: Home Heating State Machine for the Boiler Controller

The Boiler Controller State Machine in Figure 9 describes the states in which the controlled element (the Boiler) can be and the transitions between these states. Taking a transition depends on variables belonging to elements.

In this state machine the Boiler can be in the state 'NotWarmingUp' or in the state 'WarmingUp'. When state 'WarmingUp' is entered a timer is started for the Boiler. When leaving the state due to following one of the outgoing transitions, the timer is stopped.

- The transition from the 'Initial' state to the state 'NotWarmingUp' is performed without any restrictions.
- The guard of the transition from 'NotWarmingUp' to 'WarmingUp' is *true* when the temperature of the Boiler Temperature Sensor is less or equal than the low set point of the Boiler requested temperature. As an effect of the transition function 'DoWarmUp()' belonging to the Boiler is called and afterwards the 'StartTemperature' of the Boiler is set to the current temperature of the Boiler Temperature Sensor.
- The transition from 'WarmingUp' to 'WarmingUp' is enabled when the elapsed time of the boiler timer is greater or equal than a certain value called 'CheckPeriod'. The effect of this transition is then that the percentage heat needed for the Boiler is set to the value of the high set point of the Boiler minus the temperature of the Temperature Sensor, divided by the start temperature.
- The transition from 'WarmingUp' to 'NotWarmingUp' is enabled when the Boiler temperature is greater or equal than the high set point of the Boiler requested temperature. The effect is that the function 'DontWarmUp()' is called.

HeatController

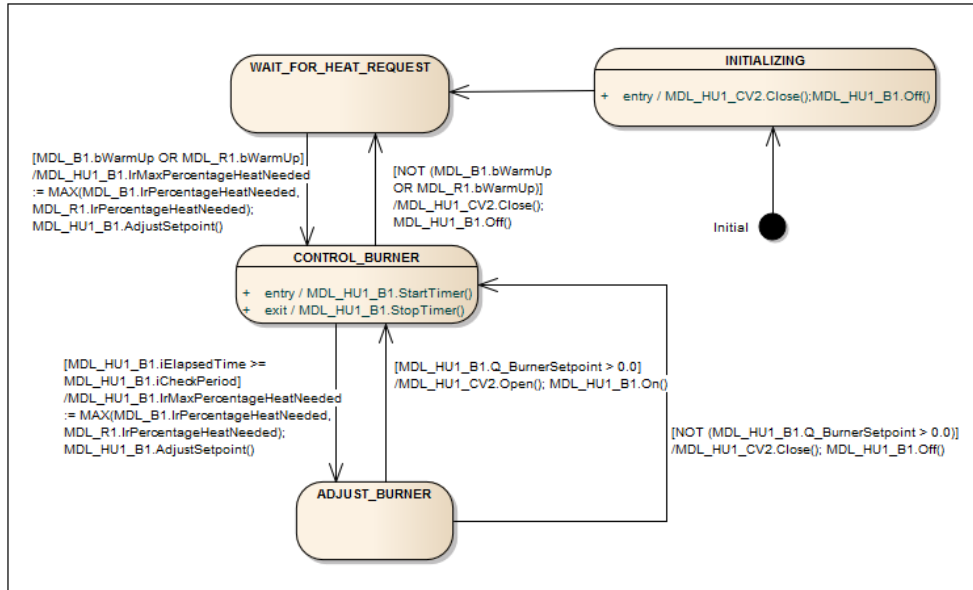


Figure 10: Home Heating State Machine for the Heat Controller

The Heat Controller State Machine in Figure 10 describes the heating states of the system, i.e., the states of the Burner. The states and transitions depend on the states of the Boiler and the Radiator.

The states the Burner can be in are INITIALIZING, WAIT_FOR_HEAT_REQUEST, CONTROL_BURNER and ADJUST_BURNER. When the INITIALIZING state is entered, the Gas valve is closed and the Burner is switched off. In state CONTROL_BURNER a timer is started when entered, and when the state is left the timer is stopped.

- From the 'Initial' state a transition is taken to state INITIALIZING.
- From the INITIALIZING state a transition is taken to state WAIT_FOR_HEAT_REQUEST.
- To go from state WAIT_FOR_HEAT_REQUEST to state CONTROL_BURNER the Boiler or the Radiator should be warmed up. The effect of the transition is then that the maximum percentage heat needed for the Burner is determined by taking the maximum of the percentages of heat needed by the Boiler and the Radiator. After that the set point of the Burner are adjusted in the function 'AdjustSetpoint()'.
- The transition going back from CONTROL_BURNER to state WAIT_FOR_HEAT_REQUEST is when there is no need to warm up the Boiler and the Radiator anymore. The effect is that the Gas valve is closed and the Burner is switched off.
- The transition to go from state CONTROL_BURNER to state ADJUST_BURNER is enabled when the elapsed time for the Burner is greater or equal than the 'CheckPeriod' value. The maximum percentage heat needed is updated and the set point adjusted in the same way as is done as effect of the transition from state WAIT_FOR_HEAT_REQUEST to state CONTROL_BURNER.
- From state ADJUST_BURNER one option to return to state CONTROL_BURNER is when the Burner set point is greater than 0, with the effect that the Gas valve is opened and the Burner turned on.
- From state ADJUST_BURNER the second option to return to state CONTROL_BURNER is when the Burner set point is smaller or equal to 0, with the effect that the Gas valve is closed and the Burner is turned off.

PumpController

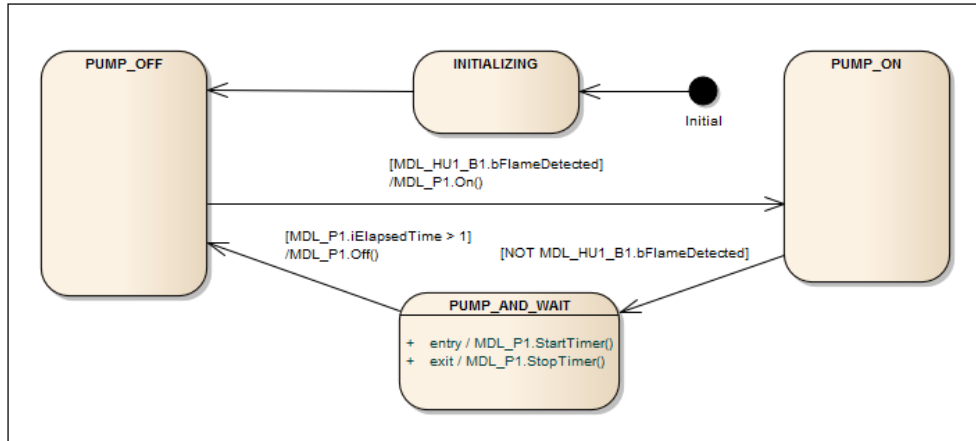


Figure 11: Home Heating State Machine for the Pump Controller

The Pump Controller State Machine in Figure 11 describes the states of the Pump.

It can be in the states INITIALIZING, PUMP_ON, PUMP_OFF or PUMP_AND_WAIT. In state PUMP_AND_WAIT a timer is started when entered and when left the timer is stopped again.

- The transition from 'Initial' to INITIALIZING is taken without restrictions.
- From the INITIALIZING state the transition has no guard or effect, so continues in the state PUMP_OFF.
- The transition from state PUMP_OFF to state PUMP_ON is enabled if there is a flame detected in the Burner. The effect is that the Pump is switched on.
- From state PUMP_ON the state PUMP_AND_WAIT is entered when there is no flame detected in the Burner.
- The transition from PUMP_AND_WAIT is taken, when the elapsed time is greater than a certain amount.
- The Pump is switched off as an effect of the transition from PUMP_AND_WAIT to PUMP_OFF.

ThreeValveController

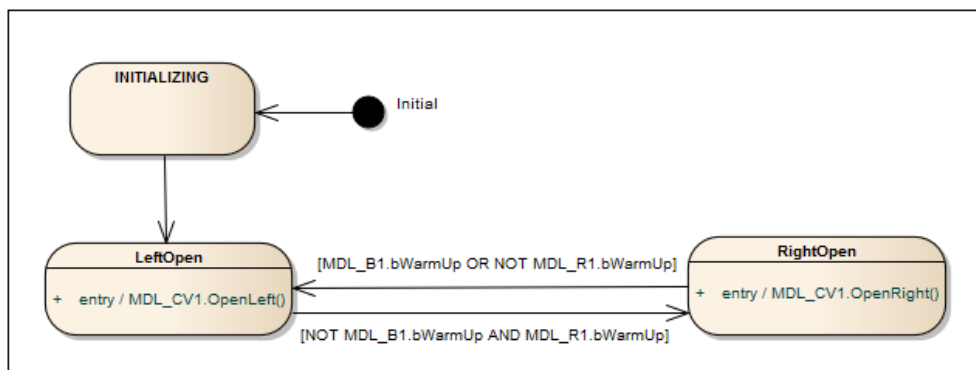


Figure 12: Home Heating State Machine for the Three Way Valve Controller

The Three Way Valve Controller State Machine in Figure 12 describes the states the three way valve can be in. The valve is positioned between the Pump, the Boiler and the Radiator.

It can be in states: INITIALIZING, 'LeftOpen' and 'RightOpen'. When the 'LeftOpen' state is entered, the Three Way Valve is opened to the left with a call to its function 'OpenLeft()'. Entering state 'RightOpen' has similar behavior.

- From the 'Initial' state a transition is taken to state INITIALIZING.
- From the INITIALIZING state a transition is taken to state 'LeftOpen'.
- When the Boiler does not need to be warmed up, but the Radiator does need to be warmed up, the transition from state 'LeftOpen' is taken to state 'RightOpen'.
- When then the Boiler does need to be warmed up or the Radiator does not need to be warmed up, state 'RightOpen' is left and the 'LeftOpen' state is entered again.

RadiatorController

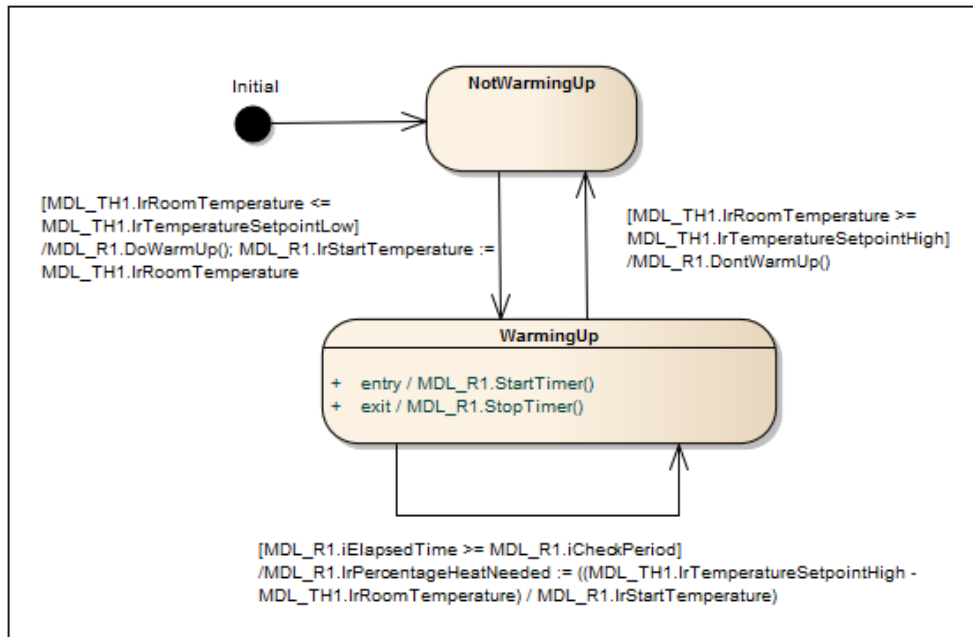


Figure 13: Home Heating State Machine for the Radiator Controller

The Radiator Controller State Machine in Figure 13 is similar to the State Machine of the Boiler Controller, since they both handle a Heat Exchanger. The difference is that the temperature sensor now is part of the Thermostat and also the requested temperature is now set by the Thermostat.

3.2 SoLayTec Case

In this Section a real life case at a company called SoLayTec is described. SoLayTec is a spin-off from TNO, a Dutch research organization. SoLayTec makes machines for Atomic Layer Deposition (ALD) on solar cells. They have a way to do that at a fast rate, to make it possible to make solar energy a competitive alternative to energy sources. The software for this machine is developed by Sioux.

The reason to use model checking in this project is that the Atomic Layer Deposition is done with gases TMA(l) (Trimethylaluminum or $Al_2(CH_3)_6$) and water vapor (H_2O). These gases are not allowed to mix, because it leads to an explosive reaction. Therefore the bearer gas N_2 is added to separate the TMA and H_2O . Figure 14 shows how the ALD Process is performed. The horizontal bar is the solar cell wafer to which the layers are added. The wafer is moved from left to right and back with an air flow, the wafer is in this way consecutively exposed to TMA and H_2O . The reaction that happens causes an atomic layer of Aluminum Oxide (Al_2O_3) to appear on the surface of the wafer. Doing this several times,

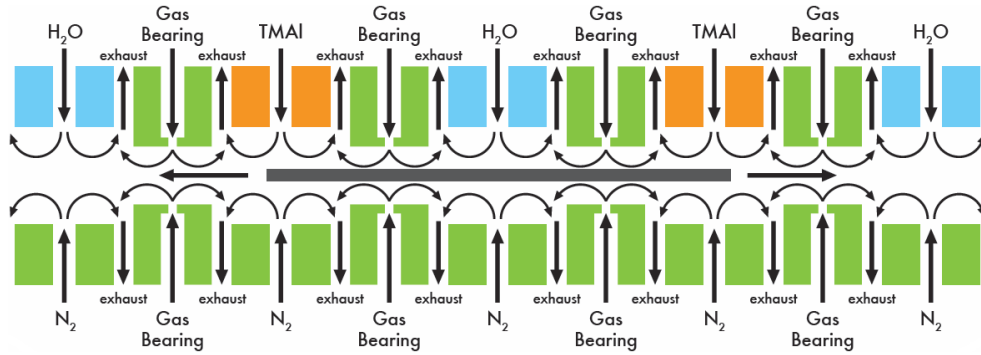


Figure 14: Atomic Layer Deposition with TMA, H₂O and N₂ (from [27])

creates a passivation layer for the solar cell wafer. This layer helps to make the solar cell more efficient.

In Section 3.2.1 the model is explained.

3.2.1 Model

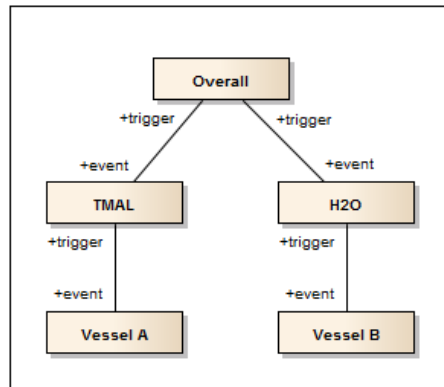


Figure 15: SoLayTec Gas test rig abstract structure

The machine contains a gas / liquid circuit, which is modeled in the Piping and Instrumentation Language. There are more elements than in the LWC Case of Section 3.1. The test rig that is built for this gas / liquid circuit is controlled by several state machines. These state machines correspond to procedures that have to be carried out by the machine. The overall hierarchical structure of the state machines is shown in Figure 15. The top node represents the overall state machine controlling the subcomponents TMAL and H₂O. In the configuration used here, they both have a subcomponent Vessel, denoting the container of the materials TMAL and H₂O. For each component in the structure there are state machines controlling the procedures for this component.

4 Translation of Models to mCRL2

For the purpose of generating an mCRL2 specification (Section 2.4) from the state machines (Section 2.3), a generator is used. To generate the correct mCRL2 code, a formal translation from the State Machine Syntax to mCRL2 code is made. This is described in Section 4.2. The translation of triggers, guards and effects as used in the state machines is done with the help of a dedicated parser described in Section 4.3. How the generation process works is explained in Section 4.4.

4.1 Translation Concept

In this section the translation from the (meta) model elements to the mCRL2 elements is described. In Table 5 the state machine elements are mapped to mCRL2 elements. The first column lists the UML State machine elements from the Meta model in Figure 5 on Page 14. The second column lists the mCRL2 syntax elements as defined in Appendix B of [12]. In the third column there are some example code snippets given that result from the translation.

When there is more than one element in the second column, this means that the element in the first column is used in different parts of the mCRL2 specification. For example a State is used when the sort of state names is defined (ConstrDecl) and also when the process expression is constructed (ProcExpr).

State machine element	mCRL2 syntax element	code example
Model	set of mCRL2Spec	
Package	set of mCRL2Spec	
Class	set of mCRL2Spec	
StateMachine	mCRL2Spec	sort Region_States = struct ...; ... act ...; ... map ... ; var ... : ...; eqn ... = ...; ... proc StateMachine(...) = ... ; ... init hide ({...}, allow ({...}, comm ({...}, StateMachine ...)));
Region	SortSpec ActSpec ProcSpec IdsDeclList ProcExpr DataExprList Init	sort Region_States = struct ...; act ...; ... proc StateMachine(...) = ... ; Region_currentState: Region_States, ... (Region_currentState == Initial) -> Initial . (...) + ... b1 && b2 if(b3,b4,b5) init hide ({...}, allow ({...}, comm ({...}, StateMachine ...)));
PseudoState	ConstrDecl ActSpec ProcExpr DataExpr	Initial act StateInitial; (Region_currentState == Initial) -> StateInitial . (...) Region_currentState == Initial
State	ConstrDecl ActSpec IdsDecl ProcExpr DataExpr	State act StateState; Region_State_currentState : Region_States (Region_currentState == State) -> StateState . (...) Region_currentState == State
FinalState	ConstrDecl ActSpec ProcExpr DataExpr	Final act StateFinal; (Region_currentState == Final) -> StateFinal . delta Region_currentState == Final
Transition	ConstrDecl ActSpec ProcExpr	guard_var act trigger; act guard; act effect; sum trigger:Bool . (trigger) -> sum guard:Bool .

State machine element	mCRL2 syntax element	code example
		(guard) -> trigger guard bool_setget_s(guard_var, guard, guard) . effect <> bool_setget_s(guard_var, guard, guard) <> tau
Constraint	Id ProcExpr	guard_var bool_setget_s(guard_var, guard, guard)
Trigger	Id ProcExpr	trigger trigger
Behavior	Id ProcExpr	effect effect

Table 5: State machine to mCRL2 mapping

The general translation goes as follows:

For each state machine in the model

- Collect all state names, initial and final states included, into the structured sort ‘States’. The names are made unique by adding a namespace, which consists of the ancestor states, so if STATE2.1 is nested in STATE2, which is part of state machine SM, the name of that state is SM.STATE2.STATE2.1. Since a model can contain state machines with the same name, the state machine names are also made unique, by adding the name of the containing class, so the full name of the state machine SM becomes CLASS.SM. Note that if the state names, must be machine parsable, the concatenation with an underscore (‘_’) must be different from what is allowed in state names.
- Collect the variable names used in the model in a structured sort ‘VarNames’.
- Collect the action names, these are trigger names, state names concatenated with ‘State’, guard and behavior variable names.
- Collect the constants and represent them as **map** and **eqn** parts.
- Make a process description with for each region a ‘currentState’ and ‘aTransitionWasPerformed’ parameter. The first is of sort ‘States’ and represents the state the corresponding region is in. The second parameter is of sort Bool and represents whether in the previous round a transition was performed.
 - In the process description, add for each region a conditional choice to check in which state the region is in. The actions to add for such a state are:
 - * If available: entry behavior. This should be wrapped in a conditional choice to check whether there was a transition performed in the previous round.
 - * If available: do behavior.
 - * If available: For each outgoing transition of the State the following is added.
 - If available: A conditional choice for the trigger.
 - If available: A conditional choice for the guard.
 - If the above conditionals are available and true and the effect is available, the effect is added. If the above conditionals are not available and the effect is available, the effect is added, also.
 - If the state is composite, translate its region, but only the part for the entry and do behavior. This is done to make sure that if a transition was performed in the previous round, all the available entry and do behavior of the entered state hierarchy is executed, before the transition is executed.
 - If available, add the exit behavior of the states exited by the transition. This is done for those states in the state hierarchy that are below the Least Common Ancestor of the source and target states.
 - Add a recursive call to the state machine with the ‘currentState’ and ‘aTransitionWasPerformed’ parameters updated.
 - * If the state is composite, translate its region in the same way as the current region is translated.

- Make a shortcut process description which is used as a shortcut for the same process representing the state machine, but then with the initial values filled in for the parameters.
- Make a coordinator process, that decides which state machine may take its turn. (LWC Design Decision 1)
- Make a process that keeps track of the variables that are used in the guards and behaviors. (LWC Design Decision 2)
- Make a process that represents the actions and effects of the environment of the system. (LWC Design Decision 5)
- Make an initial process where the environment process, the variable process, and the shortcut state machine processes are put in parallel composition.
 - To make linearizing easier, for some groups there are only the specific multi-actions allowed. Thus these have to be collected.
 - Make a communication between the variable process and the rest, so the setting and getting of variables is synchronized.
 - Depending on the purpose of the resulting mCRL2 specification, hide actions that are irrelevant.

In the above schema the ‘trigger’, ‘guard’, ‘effect’, ‘entry’, ‘do’ and ‘exit’ actions are presented too simple. They are represented by plain text, but have a meaning. This has to be taken into account and therefore parsers are made with Xtext. More on that in Section 4.3.

4.2 Formal Translation

The schema given in the Section 4.1 was fairly abstract. To be sure that the translation is correct, a formal translation is made. The input is the textual representation of the restricted UML State Machine Meta Model, which can be found in Appendix A. Since the syntax is inductive, also the translation is inductive. To deal with that in an intuitive way, a function is defined, taking a piece of UML State Machine syntax and producing mCRL2 code. For example consider the translation of a Vertex with respect to the action names in the mCRL2 specification. The textual representation of a Vertex is:

$$\begin{aligned} &Vertex(outgoing : Transition^*, incoming : Transition^*, container : Region, \\ & \qquad \qquad \qquad inherits : \{NamedElement\}). \end{aligned}$$

The translation function that corresponds is:

$$\begin{aligned} &[[Vertex(outgoing, incoming, container, inherits : \{NamedElement\})]]_{ActionName} := \\ & \quad \text{If } Vertex.metaType = Pseudostate \wedge outgoing.size > 0 \\ & \qquad \quad [[outgoing]]_{ActionName} \\ & \quad \text{ElseIf } Vertex.metaType = State \\ & \qquad \quad [[(State)this]]_{StateActionName} \end{aligned}$$

The notation $[[\cdot]]_{ActionName} :=$ denotes the function definition, with the textual representation, without the type information of the parameters, at the \cdot position. If the $[[\cdot]]_{ActionName}$ is used after the $:=$ (is defined as) sign, it is a call to a function, with at the \cdot position the parameter values. The name of the function is at the bottom right of the double square brackets. The body of the function is placed after the $:=$ sign. The If-ElseIf construction works by indentation. In the function body of this example is a distinction made on the type of the vertex, recorded in the property *metaType*. If the vertex of the input is a *Pseudostate*, i.e., an Initial state, and has at least one outgoing transition, the function *ActionName* with parameter *outgoing* is called. If the type is *State* then the input is cast to *State* and passed as parameter value to function *StateActionName*. These functions call other functions or write mCRL2 specification code.

More explanation of the notation and the translation functions for the restricted UML State Machines can be found in Appendix B.

The translation is set up in this way, so it is easily transformed to Xpand templates and Xtend helper functions. The Xpand template code for the *ActionName* function for a Vertex, as explained above is:


```

1 <<DEFINE ActionName FOR uml::Vertex->>
2     <<IF this.metaType == uml::Pseudostate && this.outgoing.size > 0->>
3         <<EXPAND ActionName FOR this.outgoing->>
4     <<ELSEIF this.metaType == uml::State->>
5         <<EXPAND StateActionName FOR (uml::State)this->>
6     <<ENDIF->>
7 <<ENDDEFINE>>

```

In this piece of Xpand the <<DEFINE ActionName FOR uml::Vertex->>... <<ENDDEFINE>> block defines the *ActionName* function, where the input is an element of type *uml :: Vertex*. This corresponds to the textual representation of a Vertex.

In the DEFINE-ENDDEFINE block the parameter element is referred to as *this*. The body of the DEFINE-ENDDEFINE block contains an IF-ELSEIF-ENDIF block, where the conditions correspond to the formal translation conditions of the If-Elseif construction. To call a function the <<EXPAND StateActionName FOR (uml::State)this->> where *StateActionName* is the function name and after the FOR keyword the parameter is given. Note that in this way only one parameter can be given to a function, but *ActionName* in the first line can be expanded with parameters too.

4.2.1 Language Workbench Challenge Design Decisions

In this section specific modeling decisions for the Language Workbench Challenge are described that influence the translation described in Sections 4.1 and 4.2.

Design Decision 1

The state machine models are designed with generation to Programmable Logic Controller code in mind. In this code a certain order is defined in which the state machines must be executed. To incorporate this order in the mCRL2 specification, the specification is extended with a Coordinator process (based on the Coordinator from [11]), which decides which state machine may execute. Most of the mCRL2 specification is generated, but the order must be defined manually.

This decision has also an impact on the state space. Since all processes (representing the state machines) are put in parallel, the state space grows exponentially in the size of the processes, but with the coordinator the state space grows linearly in the sum of the sizes of the processes. The order is of importance since this can influence the order in which certain updates or state transitions in the system can be performed.

Design Decision 2

Initially there was an mCRL2 specification generated where no data was involved. The transitions between states could be taken independent of the data playing a role on the transition. With this specification, properties where no data is involved are checked, such as reachability, but with the assumption that the data needed to get there is provided in the real system.

To verify more advanced and interesting properties, data is introduced to incorporate more detail. This detail was already part of the manual PLC code. The variables used in guards and assignments are manually defined in PLC code and the functions called, as part of for example the entry behavior in a state, are implemented there. Assignments to variables, at transitions and in states influence the behavior of the system.

To incorporate the data involved in the system, a 'Framework' class diagram is added to the Piping & Instrumentation Profile Enterprise Architect file. The Burner part of the 'Framework' is shown in Figure 16. The numbers in the figure correspond to the numbers in the description below. For each element, i.e., Boiler, Radiator, Burner, etc.:

- a class (1) is added, with for each class
 - attributes (2), representing variables and constants, with possible default and boundary values and
 - operations (3), representing the actions available for that element, such as Open and Close actions. For each operation a state machine is provided.
- an operation (4) and state machine are defined, since in PLC code, the elements function block has behavior.

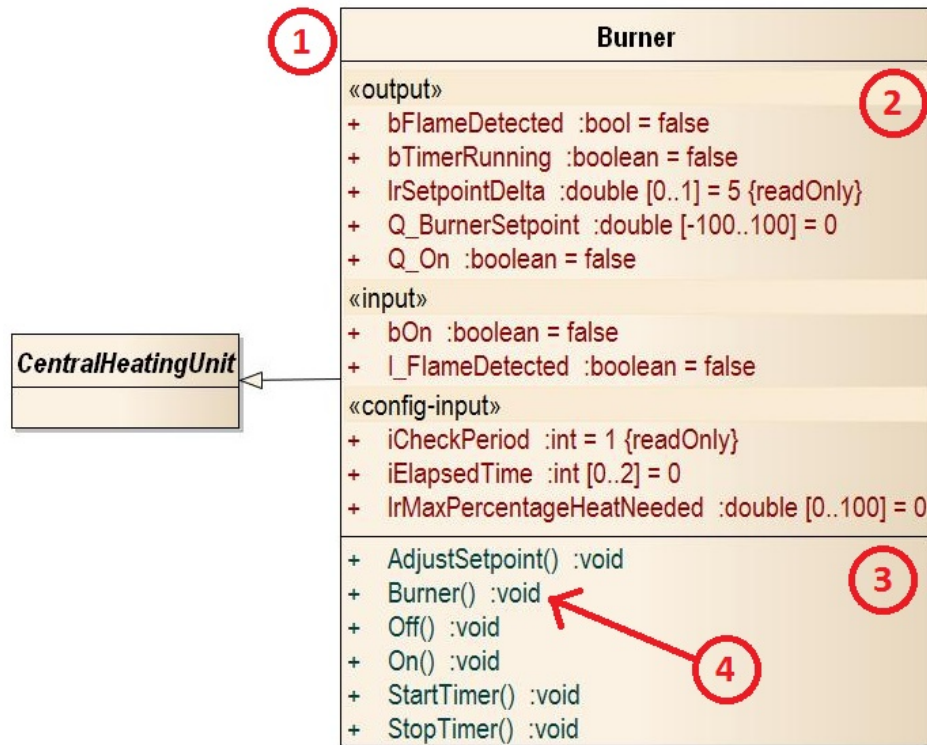


Figure 16: Burner part of the Framework of the Language Workbench Challenge Case

Default values are needed to initialize the elements. Boundary values are needed to bound the model checking, for example, the integer numbers are infinite, but when they are bounded, this yields a finite range. Constant values have the statement `{readOnly}` behind it.

The values of the variables are recorded by an mCRL2 process 'Var'. For each variable *variable* its parameters are *variable.name* ++_value, *variable.name* ++_lowerbound and *variable.name* ++_upperbound, representing the current value, lower bound and upper bound. For boolean variables, the upper and lower bounds are omitted. The ++ notation, denotes concatenation, so the value of *variable.name* is concatenated to a string, yielding another string. The process contains for every used combination of variables a summand. In the following definition the following shorthands are used:

```

name := variable.name
element := element.name
fullName := element ++name
var := name ++_var
currentValue := fullName ++_value
newValue :=new_fullName
upperBound := fullName ++_upperbound
lowerBound := fullName ++_lowerbound
Type := variable.type
type := LowerCase(variable.type)
  
```

For a boolean variable the summand is as follows:

```

sum ++ newValue ++: ++ Type ++ . ++ type ++_setget_r(++ element ++, ++ var,
++ currentValue ++, ++ newValue ++) . Var(++ currentValue ++ = ++ newValue ++)
  
```

This results for a variable *bWarmUp* of element *B1* in the following mCRL2 code:

```

sum new_B1_bWarmUp:Bool . bool_setget_r(B1, bWarmUp_var, B1_bWarmUp_value,
new_B1_bWarmUp) . Var(B1_bWarmUp_value = new_B1_bWarmUp)
  
```

For an integer or real value, the bounds are used. At the end of the first line comes:

```
(++lowerBound++ <= ++newValue++ && ++newValue++ <= ++upperBound++) -> ++
```

For the variable *iElapsedTime* of element *R1* this becomes:

```
sum new_R1_iElapsedTime:Int .
  (R1_iElapsedTime_lowerbound <= new_R1_iElapsedTime &&
   new_R1_iElapsedTime <= R1_iElapsedTime_upperbound) ->
  int_setget_r(R1, iElapsedTime_var, R1_iElapsedTime_value,
               new_R1_iElapsedTime) . Var(R1_iElapsedTime_value = new_R1_iElapsedTime)
```

The `..._setget_r` actions correspond to `..._setget_s` actions which are used for the variable occurrence in the model, so for the *iElapsedTime* variable above, the getting of the value results in:

```
sum R1_iElapsedTime:Int . int_setget_s(R1, iElapsedTime_var,
                                       R1_iElapsedTime, R1_iElapsedTime)
```

for setting the value, the last parameter of the `..._setget_s` action must have a possible different value. This value is based on the current value, but also constants or other variables are used. An example of the first case is the update of the *iElapsedTime* variable, by increasing it by 1:

```
sum R1_iElapsedTime:Int . int_setget_s(R1, iElapsedTime_var,
                                       R1_iElapsedTime, R1_iElapsedTime + 1)
```

Design Decision 3

When the mCRL2 specification with data is generated, this leads to a lot of more states in the labeled transition system it represents. The number of states became so large that model checking could not be done in a reasonable amount of time and space. Therefore several minimization techniques have been applied to reduce the state space.

hide actions With this technique the hidden actions become internal actions. These can later on be removed or used during the reduction of the state space.

put actions together to multi-actions When several actions are put sequentially executed as in the process `a . b . c` this process has four states, while the process with the multi-action `a|b|c` has only two states. If this is applied to many actions the reduction is significant.

confluence reduction This technique removes confluent τ s, i.e., internal actions. Confluence occurs when processes are put in parallel and perform their actions independently. So there are several ways to reach the same state. This phenomenon is the cause of the state space explosion problem. When τ -prioritization is used, i.e., in every state a confluent τ can be chosen and other actions can be ignored, the state space can be reduced. For more on this reduction technique see [12].

branching-bisimulation reduction This technique can be used to reduce the state space after generating it. If a τ transition can be mimicked by zero or more τ transitions and lead to a state from which the same transitions are possible, these τ transitions can be removed. This does not preserve τ -loops. These τ -loops could occur when a system is in a state, where it can stay infinitely long, but the branching-bisimulation considers the fairness property that a this never happens and a loop is always at sometime exited.

divergence-preserving branching-bisimulation reduction This technique is the same as the branching-bisimulation reduction, but it does preserve τ -loops.

Design Decision 4

For some variables in the original problem, large bounds apply. To reduce the state space it is legitimate to lower these bounds. For example the bounds of timers: A timer could run

for 500 iterations, before a certain checkpoint, say 500, is reached. To mimic the behavior, the timer has now a range from 0 to 2 and the checkpoint is set to 1. The timer is normally increased by an assignment $timer := timer + 1$, but the way it allows this assignment is to match the new value $timer + 1$ with the bound. With $timer = 2$ this would introduce a deadlock. To overcome this, in the mCRL2 specification the statement

```
int_setget_s(<element>, <timer_var>, timer, timer + 1)
```

is replaced by

```
int_setget_s(<element>, <timer_var>, timer, if(timer == 2, 2, timer + 1)).
```

Design Decision 5

The environment, i.e., the hardware and user interaction, is modeled in the LWC solution by simulation. The simulation is implemented in simulation function blocks, corresponding to the controller function blocks (from the Framework) for the elements. This environment is also needed in the mCRL2 specification to restrict the behavior of the system. This environment is modeled by an extra process called 'Env'. This process sets variables to certain values (normally their bound values) taking the values of the current state of the system into account. With knowledge of mCRL2 this process can be modeled manually, otherwise the behavior should be modeled as a state machine as part of a Controller. When it is manually added to the mCRL2 specification, it is more efficient, because the behavior can be put in one transition, while in the state machine it is clearer to put it in several transitions.

4.2.2 SoLayTec Design Decisions

In this section the design decisions that are made for the SoLayTec project are described and explained. These design decision are an extension of the design decisions made for the LWC 2012 case in Section 4.2.1.

Design Decision 1

A Framework as described in Design Decision 2 of Section 4.2.1 is also added here to the Piping & Instrumentation Language Profile file. The variables, constants and functions are only defined for the things that influence variables that are used in the properties to check. This is done to reduce the number of variables influencing the state space, because when there are a lot of variables, they can make the state space explode.

Design Decision 2

The class with the controller state machines for the test rig is moved from the Software Architecture Model, which contains all the software models for the SoLayTec ALD Machine, to the Test Rig Model. This class with state machines is then stereotyped to Controller. This is done to be compatible with the generator workflow script created for the LWC Case.

Design Decision 3

In the controllers, global variables and constants are used in calculations of effects and in guards. Since in the model checking data variables are used in the properties, the variables and constants that influence the 'property variables' are modeled in the Controller class, which also contains the state machines. This is done to take the important variables and values into account while checking the properties involving data variables.

Design Decision 4

The constants used are written with capital letters and in the attribute definition of the class they are marked as constant. This is done to distinguish them from variables.

Design Decision 5

For the Interlock checking (see Section 5.2.3) the guards and effects are flattened, i.e., reduced to a boolean variable for the parts that do not contain the variables, used in the interlocks, or that influence them. This decision is strongly connected to Design Decision 1 of this section. This decision is made to reduce the number of variables influencing the state space.

4.3 Parser

In this section the translation of the textual representation of triggers, guards and effects that occur on transitions and in states is described. To translate these texts to mCRL2, a dedicated parser is developed. The reason to make this parser is that the triggers, guards and effects are represented by ordinary strings in the model and thus lack any meaning. By providing a parser for this, the string gets meaning.

The grammars as described in Sections 4.3.1, 4.3.2 and 4.3.3 are the basis for the parser. They are coded in Xtext, a formalism to describe a language. With this grammar description a parser is generated. The Eclipse project in which this is done was then exported to a deployable plug-in, which is used in the generator project as a library. For each trigger, guard or effect passed as string to the parser a structured representation of the parsed string is returned. This representation is traversed by some Java functions, to generate an mCRL2 string. The parser is divided into three different sub parts. To distinguish the strings between those sub parts, the prefixes ‘trigger’ for triggers, ‘constraint’ for guards and ‘effect’ for effects are added to the strings to be parsed.

In the coming sections, there are first the (lexical) grammar rules defined and after that the formal translation into mCRL2. The parsing happens on the fly, so the structure of the text is not available for Xpand to traverse over. Therefore the translation is not done by Xpand templates, but with Java code.

The grammar notation can be read as follows: the right hand side of the grammar rules (after ::=) contains bracket pairs “(..)” to denote grouping, the “*” notation means zero or more times, the “+” notation means one or more times and the “?” notation means zero or one times. Furthermore the “..” notation represents a range. In the grammar there is distinction between normal production rules and lexical elements. The lexical elements specify the characters of the building blocks that can be used by the production rules. For example *ID* is a lexical element and is for example used as a building block in the production rule *Field*, which is either an *ID* followed by any number of a dot and an *ID*, or a *CAPITAL* followed by one or more dot and *ID* pairs. Thus Fields that are parsed correctly are `field`, `field.param1.param2` and `ELEMENT.param1`, but `ELEMENT` is not a correct Field.

4.3.1 Trigger Parser

Grammar

A trigger has the following grammar rules:

$$\begin{aligned} Model &::= \text{“trigger” } Field; \\ Field &::= ID(\text{“.”}ID)^* \mid CAPITAL(\text{“.”}ID)^+; \end{aligned}$$

Lexical elements

$$\begin{aligned} ID &::= (\text{‘^’})?(\text{‘a’}..\text{‘z’}|\text{‘A’}..\text{‘Z’}|\text{‘_’})(\text{‘a’}..\text{‘z’}|\text{‘A’}..\text{‘Z’}|\text{‘_’}|\text{‘0’}..\text{‘9’})^*; \\ CAPITAL &::= (\text{‘_’}|\text{‘A’}..\text{‘Z’})(\text{‘_’}|\text{‘A’}..\text{‘Z’}|\text{‘0’}..\text{‘9’})^*; \end{aligned}$$

Translation

The translation of a trigger parsed by the parser based on the above grammar rules is:

$$\begin{aligned} \llbracket trigger \rrbracket_{Model} &:= \\ &\llbracket (Field)trigger \rrbracket_{Field} \\ \llbracket field \rrbracket_{Field} &:= \\ &\text{If } field.fields.size > 0 \\ &\quad field.name \text{ ++ } \llbracket field.fields \rrbracket_{FieldParams} \\ &\text{Else} \\ &\quad field.name \\ \llbracket field \triangleright fields \rrbracket_{FieldParams} &:= \\ &\text{If } fields.size > 0 \\ &\quad _ \text{ ++ } field \text{ ++ } \llbracket fields \rrbracket_{FieldParams} \\ &\text{Else} \\ &\quad _ \text{ ++ } field \end{aligned}$$

4.3.2 Constraint Parser

Grammar

A guard has the following grammar rules:

$$\begin{aligned}
Model &::= \text{“constraint” } OrBool; \\
OrBool &::= XorBool(\text{‘OR’} OrBool)?; \\
XorBool &::= AndBool(\text{‘XOR’} XorBool)?; \\
AndBool &::= NotBool(\text{‘AND’} AndBool)?; \\
NotBool &::= (\text{‘NOT’})? BoolExpr; \\
BoolExpr &::= Add(Op BoolExpr)?; \\
Add &::= Sub(\text{‘+’} Add)?; \\
Sub &::= Times(\text{‘-’} Sub)?; \\
Times &::= Divide(\text{‘*’} Times)?; \\
Divide &::= Min(\text{‘/’} Divide)?; \\
Min &::= (\text{‘-’})? Bracket; \\
Bracket &::= (\text{‘(’} OrBool\text{‘)’} | Const); \\
Const &::= \text{“TRUE”} | \text{“FALSE”} | Field | VarFunc | INT | REAL | \\
&\quad CAPITAL; \\
VarFunc &::= Function | MinMax | FunctionUpdate; \\
Function &::= (ID | CAPITAL)(\text{‘.’} ID) * (\text{‘(’} OrBool(\text{‘.’} OrBool)*\text{‘)’}?)?; \\
MinMax &::= \text{“MIN(”} OrBool(\text{‘.’} OrBool)\text{“+”} | \\
&\quad \text{“MAX(”} OrBool(\text{‘.’} OrBool)\text{“+”} | \\
FunctionUpdate &::= (ID | CAPITAL)(\text{‘.’} ID)*(\text{‘Assignment’}); \\
Op &::= \text{‘=’} | \text{‘<>’} | \text{‘<’} | \text{‘>’} | \text{‘<=’} | \text{‘>=’};
\end{aligned}$$

Lexical elements

$$\begin{aligned}
INT &::= (\text{‘-’})?(\text{‘0’}..\text{‘9’})^+ \\
REAL &::= (INT\text{‘.’}INT) | (INT\text{‘E’}(\text{‘-’})?INT);
\end{aligned}$$

Translation

The translation of the above syntax elements is:

$$\begin{aligned}
[[constraint]]_{Model} &:= \\
&\quad [[(OrBool)constraint]]_{OrBool} \\
[[orbool]]_{OrBool} &:= \\
&\quad [[orbool.lhs]]_{XorBool} ++ || ++ [[orbool.rhs]]_{OrBool} \\
[[xorbool]]_{XorBool} &:= \\
&\quad (++ [[xorbool.lhs]]_{AndBool} ++ \&\& ! ++ [[xorbool.rhs]]_{XorBool} ++) || ++ \\
&\quad (! ++ [[xorbool.lhs]]_{AndBool} ++ \&\& ++ [[xorbool.rhs]]_{XorBool} ++) \\
[[andbool]]_{AndBool} &:= \\
&\quad [[andbool.lhs]]_{NotBool} ++ \&\& ++ [[andbool.rhs]]_{AndBool} \\
[[notbool]]_{NotBool} &:= \\
&\quad \text{If } notbool.is_not \\
&\quad \quad ! ++ [[notbool.boolexpr]]_{BoolExpr} \\
&\quad \text{Else} \\
&\quad \quad [[notbool.boolexpr]]_{BoolExpr} \\
[[boolexpr]]_{BoolExpr} &:= \\
&\quad [[boolexpr.lhs]]_{Add} ++ [[boolexpr.op]]_{Op} ++ [[boolexpr.rhs]]_{BoolExpr} \\
[[add]]_{Add} &:=
\end{aligned}$$

```

[[add.lhs]]Sub ++ + ++ [[add.rhs]]Add

[[sub]]Sub :=
  [[sub.lhs]]Times ++ - ++ [[sub.rhs]]Sub

[[times]]Times :=
  [[times.lhs]]Divide ++ * ++ [[times.rhs]]Times

[[divide]]Divide :=
  [[divide.lhs]]Min ++ / ++ [[divide.rhs]]Divide

[[min]]Min :=
  If min.is_min
    -++ [[min.bracket]]Bracket
  Else
    [[min.bracket]]Bracket

[[bracket]]Bracket :=
  If bracket.brackets
    ( ++ [[bracket.orbool]]OrBool ++ )
  Else
    [[bracket.const]]Const

[[const]]Const :=
  If const = 'TRUE'
    true
  ElseIf const = 'FALSE'
    false
  ElseIf const.isField
    [[(Field)const]]Field
  ElseIf const.isVarFunc
    [[(VarFunc)const]]VarFunc
  Else
    const

[[varfunc]]VarFunc :=
  If varfunc.isFunction
    [[(Function)varfunc]]Function
  ElseIf varfunc.isMinMax
    [[(MinMax)varfunc]]MinMax
  ElseIf varfunc.isFunctionUpdate
    [[(FunctionUpdate)varfunc]]FunctionUpdate

[[function]]Function :=
  If function.params.size > 0
    function.name ++ [[function.fields]]FieldParams ++ ( ++ [[function.params]]Params ++ )
  Else
    function.name ++ [[function.fields]]FieldParams

[[orbool > orbools]]Params :=
  If orbools.size > 0
    [[orbool]]Orbool ++ , ++ [[orbools]]Params
  Else
    [[orbool]]Orbool

[[minmax]]MinMax :=
  If minmax.is_min
    min( ++ [[minmax.params]]Params ++ )
  Else
    max( ++ [[minmax.params]]Params ++ )

```

```

[[functionupdate]]FunctionUpdate :=
  If functionupdate.fields.size > 0
    functionupdate.name[[functionupdate.fields]]FieldParams ++
      ( ++ [[functionupdate.ass]]Assignment ++ )
  Else
    functionupdate.name ++ ( ++ [[functionupdate.ass]]Assignment ++ )

[[op]]Op :=
  If op = '='
    ==
  ElseIf op = '<>'
    !=
  Else
    op

```

4.3.3 Effect Parser

Grammar

An effect often consists of more than one effect, thus lists of effects are considered. A list of effects has the following syntax:

```

Model ::= "effect" Effect(';')?(Effect(';')?)*;
Effect := VarFunc | Assignment;
Assignment := Field2":=" (OrBool | STRING);
Field2 ::= ID(':ID')* | CAPITAL(':ID')*;

```

Lexical Element

```

STRING ::= "" ( '\\'('b' | 't' | 'n' | 'f' | 'r' | 'u' | "" | "''") | '\\') | !('\\" | "'') ) * "" |
"" ( '\\'('b'|'t'|'n'|'f'|'r'|'u'|""|'""'|'\\') | !('\\"|'""') ) * "";

```

This just means any plain text within “double” or ‘single’ quotes.

Translation

The translation of the above syntax elements is:

```

[[effects]]Model :=
  [(List[Effect])effects]Effects

[[effect▷effects]]Effects :=
  If effects.size > 0
    [[effect]]Effect ++ . ++ [[effects]]Effects
  Else
    [[effect]]Effect

[[effect]]Effect :=
  If effect.is_VarFunc
    [(VarFunc)effect]VarFunc
  Else
    [(Assignment)effect]Assignment

[[assignment]]Assignment :=
  If assignment.rhs.is_OrBool
    [[assignment.lhs]]Field2 ++ _to_ ++ [(OrBool)assignment.rhs]OrBool
  Else
    [[assignment.lhs]]Field2 ++ _to_ ++ assignment.rhs

[[field2]]Field2 :=
  If field2.params.size > 0
    field2.name ++ [[field2.params]]FieldParams
  Else
    field2.name

```


4.4 Generator overview

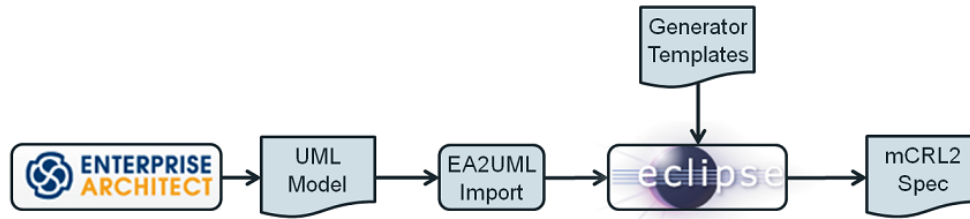


Figure 17: The translation toolchain

The mCRL2 generation process is based on the generation process for PLC code from state machines as used at Sioux. It starts with a UML Model (and UML Meta Model or Profile) made in Enterprise Architect (EA) [29]. The EA file format is not directly usable by the generator in Eclipse [8], therefore the file is translated to a UML format which is usable. This is done with the EA2UML library [6], which transforms the EA file to an XML file. This format has a hierarchical structure, and is thus very much suitable to traverse over. This can be easily done with the openArchitectureWare / Eclipse modeling Project tooling [24, 9]. This tooling consists of the components Workflow Engine, Xpand, Xtend, Check and Xtext, with Java as programming language behind it.

With the Workflow Engine workflow scripts are executed. The workflow script for the generation process, manages the translation from the EA file to the usable XML format and after that the translation from that XML format to an mCRL2 Specification.

The actual generation of the mCRL2 code, is done using Xpand templates. These templates use functions defined in the Xtend language. For more complex functions a call to Java functions are made. The Check language, for checking some constraints on the model, is not really used, except for the parts which were already present in the PLC generator. The Xtext language is used to describe parser rules. With a dedicated workflow script a parser and accompanying editor are generated. In this project only the parser is used.

Figure 17 captures the above description of the tool chain visually.

5 Verification

The goal of the translation of UML State Machines to an mCRL2 Specification is formal verification. This is done with the mCRL2 toolset (version 201202.0.10804 (Release)) and the LTSMIN toolset (version `ltsmin-1.8-dirty`). In Appendix C a description of the used tools can be found. For the formal verification it is not enough to have only an mCRL2 Specification. It is also needed to have properties or requirements to which the modeled system should adhere. For each property an explanation of the verification is given. If the property holds, a proof is given and if the property does not hold, an explanation is given why it does not hold. The properties are divided into liveness and safety properties. The properties for the LWC 2012 case are captured in Section 5.1 and the properties for the SoLayTec case can be found in Section 5.2.

In the coming sections command line tools are used and the commands are preceded by a `$` sign, denoting the command prompt. With the following command a Linear Process Specification (LPS) `model.lps` is generated from an mCRL2 Specification `model.mcr12`. This specification was obtained by running the generator workflow script with the files and parameters corresponding to the desired project.

```
$ mcr122lps model.mcr12 | lpsconstelm | lpsparelm | lpssuminst | lpsconstelm |
  lpsparelm > model.lps
```

With this command `mcr122lps` takes an mCRL2 Specification and makes a Linear Process Specification (LPS) of it. From this LPS constants are removed with `lpsconstelm` and then parameters with `lpsparelm`. Summands are instantiated, so for example `sum b:Bool . a(b)` is rewritten to `a(true) + a(false)`. Then again constants and parameters are removed. The resulting LPS is used directly or translated to a Labeled Transition System (LTS), reduced modulo some equivalence relation and then converted back to an LPS. This is done with the following commands:

```
$ lps2lts -rjittyc --alternative --cached --prune model.lps model.aut
$ ltsconvert -edpbranching-bisim model.aut model_dpbb.aut
$ lts2lps -lmodel.lps model_dpbb.aut model_dpbb.lps
```

In the first line the options are to do the translation with the compiled Just In Time rewrite strategy (the compiled version is not available under Windows), making use of the alternative implementation together with caching and pruning. The ‘aut’ extension is used so the output is immediately written, which saves time in the total process. The transition information is kept, but the state information is lost, but this can be solved to add actions that have data parameters that represent the state values. But in most cases the state information is not needed.

In the second line the conversion is done with the divergence-preserving branching-bisimulation equivalence relation.

In the third line the LTS is transformed back to an LPS. This LPS can then be used for the model checking.

The actions used in the natural language and μ -calculus properties are of the form

HU1-CV2.Open().

In the state machine models of Section 3.1 the action is represented by

MDL_HU1-CV2.Open()

and in machine readable μ -calculus format (and mCRL2 format) it is denoted as

`actControlledValve_Open(MDL_HU1-CV2)`.

5.1 Language Workbench Challenge Requirements

In this section the requirements or properties to which the Home Heating System should adhere are specified and verified.

5.1.1 Liveness properties

Recall from Section 2.5 that a liveness property states that something good will happen in the future.

Property 1: Reachability

For each action that should occur in the specification, a μ -calculus formula is given. Let A be the set of actions that should occur, then for each *action* in A must hold:

$$\langle true^* \cdot action \rangle true \quad (1)$$

meaning ‘there exists a path on which *action* occurs’. In this case $A = \{HU1_CV2.Open(), HU1_CV2.Close(), HU1_B1.On(), HU1_B1.Off(), CV1.OpenLeft(), CV1.OpenRight(), B1.DoWarmUp(), B1.DontWarmUp(), R1.DoWarmUp(), R1.DontWarmUp(), P1.Off(), B1.StartTimer(), R1.StartTimer(), HU1_B1.StartTimer(), P1.StartTimer(), P1.On(), B1.StopTimer(), R1.StopTimer(), HU1_B1.StopTimer(), P1.StopTimer()\}$

These 20 formulas are all true, meaning there exists a path from the initial state to a state where these actions can be performed. These formulas could be checked with `lps2pbes` and `pbes2bool`, but since the model with data is very large, this can take very long. Also when using `lps2lts` with the option to look for particular actions while generating the Labeled Transition System takes often a very long time. The solution to this was making use of the `ltsmin` toolset, in which a tool called `lps-reach` is present, which was run with the following command:

```
$ lps-reach -rgs --mcrl2-readable-edge-labels --action='<action>' model.lps
```

where the options have the following meaning:

- `-rgs` This is an option to enable regrouping of the next state dependency matrix with transformation `gs` (Group Safely: group columns, group rows, swap columns and sort rows).
- `--mcrl2-readable-edge-labels` This is an mCRL2 option to use human readable edge labels, without this option set, the action will not be found.
- `--action='<action>'` The option to detect an action. The name of the action should be filled in instead of `<action>`. Note that the action name should be the name including the data parameters.

When the action is found it returns:

```
lps-reach: found action <action>
lps-reach, ** error **: exiting now
```

In most cases the action was found in a few seconds.

Property 2: Warming Up

This property is explained in full detail, this is done to provide insight into the process of verification and property refinement.

A requirement for the system is that the Three Way Valve must be opened to the correct side, namely to the left if the Boiler needs to be warmed up and to the right if the Radiator needs to be warmed up. For the Boiler this was first stated in the following property: If Boiler $B1$ needs to be warmed up, Three Way Valve $CV1$ needs to be opened to the left.

$$[true^* \cdot B1.DoWarmUp()](true^* \cdot (CV1.OpenLeft() \cup CV1.OpenBoth()))true \quad (2)$$

With the command:

```
$ lps2pbes -f34.mcf model.lps | pbes2bool -rjittyc -s3
```

the verification of this formula returns *true*, because after each action $B1.DoWarmUp()$ a trace exists on which $CV1.OpenLeft()$ occurs, because that is actually what the formula says: ‘whenever a $B1.DoWarmUp()$ action has occurred, there is a path on which $CV1.OpenLeft()$ or $CV1.OpenBoth()$ occurs’. This statement is too weak, because paths exist that possibly do not have a $CV1.OpenLeft()$ or $CV1.OpenBoth()$ action on it. A stronger statement is the following:

$$[true^* \cdot B1.DoWarmUp()]\mu X. \overline{[CV1.OpenLeft() \cup CV1.OpenBoth()]} X \wedge \langle true \rangle true \quad (3)$$

stating that ‘whenever a $B1.DoWarmUp()$ action has occurred, a $CV1.OpenLeft()$ or $CV1.OpenBoth()$ action is inevitably done’. Verifying this formula with the command:

`$ lps2pbes -f35.mcf model.lps | pbes2bool -rjittyc -s3 -c`

the tool returns *false*. The counterexample returns a trace, which, with simulation in `lpsxsim`, returns a path which contains an infinite loop on which `CV1.OpenLeft()` does not occur. This is caused by the fact that the Three Way Valve was already opened to the left, and thus does not need to be opened to the left if the Boiler needs to be warmed up. When also the Manual Valve is open, the Boiler stays in this state and thus never comes in a state where the Three Way Valve is opened to the left.

The outcome of formula 3 gives rise to another formula, incorporating the fact that before a `B1.DoWarmUp()` action already a `CV1.OpenLeft()` action could have taken place:

$$\begin{aligned}
& \nu X(bOL : \mathbb{B} = false). \\
& \quad [CV1.OpenRight()]X(false) \wedge \\
& \quad [CV1.OpenLeft() \cup CV1.OpenBoth()]X(true) \wedge \\
& \quad \overline{[CV1.OpenLeft() \cup CV1.OpenBoth() \cup CV.OpenRight()]}X(bOL) \wedge \\
& \quad [B1.DoWarmUp()](\\
& \quad \quad bOL \vee \mu Y.(\\
& \quad \quad \overline{[CV1.OpenLeft() \cup CV1.OpenBoth()]}Y \wedge \langle true \rangle true)
\end{aligned} \tag{4}$$

The formula starts with a νX fixed point, meaning that it starts an infinite search for the actions that are in its body. While searching and finding, a variable (*bOL*) records whether the Three Way Valve is open to the left. In the second line, the action `CV1.OpenRight()` makes this variable *false* and the action `CV1.OpenLeft()` or `CV1.OpenBoth()` makes the variable *true*. If an action other than these three are encountered the variable does not change.

When the search encounters a `B1.DoWarmUp()` action, it checks whether the value of the variable is *true* or continues a finite search for a `CV1.OpenLeft()` or `CV1.OpenBoth()` action.

This formula gives *true* as result. For the Radiator the verification is similar as for the Boiler. The used formulas are:

$$[\overline{true^* \cdot R1.DoWarmUp()}] \langle true^* \cdot (CV1.OpenRight() \cup CV1.OpenBoth()) \rangle true \tag{5}$$

$$[\overline{true^* \cdot R1.DoWarmUp()}] \mu X. \overline{[CV1.OpenRight() \cup CV1.OpenBoth()]}X \wedge \langle true \rangle true \tag{6}$$

Results

The results are: *true* (5) and *false* (6).

The explanation of Formula 6 being false is as follows: The Boiler can ask for heat indefinitely, in the case that the Manual Valve is open. In this case the Three Way Valve is opened to the left. The Radiator asking for heat is never served, because the state machine transition which is guarded by $\neg B1.bWarmUp \wedge R1.bWarmUp$, representing that only the Radiator is asking for heat, becomes never true.

Formula 6 gives rise to another formula in the same way as for Formula 3, but then with Left and Right exchanged:

$$\begin{aligned}
& \nu X(bOR : \mathbb{B} = false). \\
& \quad [CV1.OpenLeft()]X(false) \wedge \\
& \quad [CV1.OpenRight() \cup CV1.OpenBoth()]X(true) \wedge \\
& \quad \overline{[CV1.OpenLeft() \cup CV1.OpenBoth() \cup CV.OpenRight()]}X(bOR) \wedge \\
& \quad [R1.DoWarmUp()](\\
& \quad \quad bOR \vee \mu Y.(\\
& \quad \quad \overline{[CV1.OpenRight() \cup CV1.OpenBoth()]}Y \wedge \langle true \rangle true)
\end{aligned} \tag{7}$$

Formula 7 returns *false*. From the counterexample it could be inferred that the same problem as with Formula 6 occurs. To deal with that, the formula needs to incorporate also that between the `R1.DoWarmUp()` action and inevitably doing a `CV1.OpenRight()` or `CV1.OpenBoth()` action, there must have been a `B1.DontWarmUp()` action, making the `B1.bWarmUp` variable false and thus enabling the transition from the state ‘LeftOpen’ to

state ‘RightOpen’ in the ThreeWayValveController state machine diagram (See Figure 12 on Page 26). The formula representing this is:

$$\begin{aligned}
& \nu X(bOR : \mathbb{B} = false, bBD : \mathbb{B} = false). \\
& \quad [CV1.OpenLeft()]X(false, bBD) \wedge \\
& \quad [CV1.OpenRight() \cup CV1.OpenBoth()]X(true, bBD) \wedge \\
& \quad [B1.DoWarmUp()]X(bOR, true) \wedge \\
& \quad [B1.DontWarmUp()]X(bOR, false) \wedge \\
& \quad \overline{[CV1.OpenLeft() \cup CV1.OpenRight() \cup CV1.OpenBoth() \cup \\
& \quad B1.DoWarmUp() \cup B1.DontWarmUp()]X(bOR, bBD) \wedge \\
& \quad [R1.DoWarmUp()](bOR \vee \nu Z(b : \mathbb{B} = bBD). \mu Y. \\
& \quad \quad (-b \Rightarrow [CV1.OpenRight() \cup CV1.OpenBoth() \cup B1.DoWarmUp()])Y \wedge \\
& \quad \quad \langle true \rangle true) \wedge \\
& \quad \quad [B1.DoWarmUp()]Z(true) \wedge \\
& \quad \quad [B1.DontWarmUp()]Z(false) \wedge \\
& \quad \quad \overline{[CV1.OpenRight() \cup CV1.OpenBoth() \cup B1.DoWarmUp() \cup \\
& \quad \quad B1.DontWarmUp()]}Z(b)} \quad (8)
\end{aligned}$$

In this formula some things are recorded by data parameters added to some fixed point variables: *bOR* records whether the Three Way Valve is opened to the right, *bBD* and *b* record whether the Boiler needs to be warmed up. The formula starts with a greatest fixed point, so an infinite search is started. By reaching certain actions, the variables are updated and the search continues.

If *CV1.OpenLeft()* is reached then the variable *bOR* is set to *false* and if *CV1.OpenRight()* or *CV1.OpenBoth()* is reached then the variable *bOR* is set to *true*.

If *B1.DoWarmUp()* is reached then the variable *bBD* is set to *true* and if *B1.DontWarmUp()* is reached then the variable *bBD* is set to *false*.

If an action different from *CV1.OpenRight()*, *CV1.OpenLeft()*, *CV1.OpenBoth()*, *B1.DoWarmUp()* or *B1.DontWarmUp()* is reached the search is continued with unchanged variables.

If *R1.DoWarmUp()* is reached, there are two cases which makes the formula true:

- variable *bOR* is true, meaning that the Three Way Valve is open to the right, or
- variable *bOR* is false, meaning that the Three Way Valve is not open to the right and the search should continue to look for a state where the Three Way Valve is opened to the right.

The formula first used to achieve this was:

$$\mu Y. \overline{[CV1.OpenRight() \cup CV1.OpenBoth()]}Y \wedge \langle true \rangle true$$

This formula searches for a finite path on which *CV1.OpenRight()* or *CV1.OpenBoth()* occurs, but there is a cyclic path on which those actions do not occur.

This can be solved by wrapping this formula in a ν -formula, searching for an infinite path, where

- the Boiler needs to be warmed up forever, as in the cyclic path mentioned before (which is allowed behavior), or
- the infinite path is broken by the action *B1.DontWarmUp()*, enabling the search for a state where *CV1.OpenRight()*, *CV1.OpenBoth()* or *B1.DoWarmUp()* can be done. Action *B1.DoWarmUp()* is allowed, because it can indicate the start of the cyclic path mentioned before.

To mark which search should be active, depending on the Boiler’s need to be warmed up, the variable *b* was introduced. If *b* is *true*, marking that the Boiler needs to be warmed up, the formula searches for a path on which it does not need to be warmed up anymore and if *b* is *false*, the formula searches for a path that opens the Three Way Valve to the right or where the Boiler needs to be warmed up again.

Assuming that the variables *bOR* and *bBD* are initially *false*, verification of this formula yields *true*.

Property 3: Not Warming Up

When the Burner *HU1_B1* has been turned On, then at some moment the Boiler *B1* or Radiator *R1* do not need to be warmed up. The first formula used to verify this was:

$$\langle true^* \cdot HU1_B1.On() \rangle \mu X. \overline{[B1.DontWarmUp() \cup R1.DontWarmUp()]} X \wedge \langle true \rangle true \quad (9)$$

Results

Formula 9, stating that ‘after the Burner is turned on, inevitably the Boiler does not have to be warmed up or the Radiator does not have to be warmed up’, was checked with the command:

```
$ lps2pbcs -f29.mcf model.lps | pbcs2bool -rjittyc -s3 -c
```

The result it returned was *false*. The counterexample gave a trace which ended in a cycle on which the property does not hold. For this formula this is the case for the part that the Boiler or the Radiator do not need to be warmed up. Interpreting the path ending in that state, it became clear that the property was violated because the Boiler asks to be warmed up, but at the same time the Manual Valve is open and thus the Boiler is immediately using its heat, thus the Boiler staying in the same state asking for heat and delivering heat and not reaching the state where there is no heat needed anymore.

The formula is enhanced so that the counter example is excluded, because the counterexample represents desired behavior:

$$\begin{aligned} & \nu X (BDo : \mathbb{B} = false, RDo : \mathbb{B}, HS : \mathbb{B} = false). \\ & \quad [B1.DoWarmUp()] X (true, RDo, HS) \wedge \\ & \quad [B1.DontWarmUp()] X (false, RDo, HS) \wedge \\ & \quad [R1.DoWarmUp()] X (BDo, true, HS) \wedge \\ & \quad [R1.DontWarmUp()] X (BDo, false, HS) \wedge \\ & \quad [\exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j)] X (BDo, RDo, false) \wedge \\ & \quad [\exists_{b:\mathbb{B}, i, j:Z} env(900, b, i, j)] X (BDo, RDo, true) \wedge \\ & \quad \overline{[B1.DoWarmUp() \cup B1.DontWarmUp() \cup R1.DoWarmUp() \cup R1.DontWarmUp() \cup \\ & \quad \exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j) \cup \exists_{b:\mathbb{B}, i, j:Z} env(900, b, i, j)]} X (BDo, RDo, HS) \wedge \\ & \quad [HU1_B1.On()]} ((\neg BDo \wedge \neg RDo) \vee \nu Y (BDo_1 : \mathbb{B} = BDo, RDo_1 : \mathbb{B} = RDo, \\ & \quad HS_1 : \mathbb{B} = HS). \mu Z. (\\ & \quad ((BDo_1 \wedge HS_1) \Rightarrow \overline{[B1.DontWarmUp() \cup B1.DoWarmUp() \cup R1.DoWarmUp() \cup \\ & \quad \exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j)]} Z \wedge \langle true \rangle true) \wedge \\ & \quad ((\neg BDo_1 \wedge RDo_1) \Rightarrow \overline{[R1.DontWarmUp() \cup B1.DoWarmUp() \cup R1.DoWarmUp() \cup \\ & \quad \exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j)]} Z \wedge \langle true \rangle true) \wedge \\ & \quad [\exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j)] Y (BDo_1, RDo_1, false) \wedge \\ & \quad [\exists_{b:\mathbb{B}, i, j:Z} env(900, b, i, j)] Y (BDo_1, RDo_1, true) \wedge \\ & \quad [B1.DoWarmUp()] Y (true, RDo_1, HS_1) \wedge \\ & \quad [R1.DoWarmUp()] Y (BDo_1, true, HS_1) \wedge \\ & \quad \overline{[B1.DoWarmUp() \cup \\ & \quad B1.DontWarmUp() \cup R1.DoWarmUp() \cup R1.DontWarmUp() \cup \\ & \quad \exists_{b:\mathbb{B}, i, j:Z} env(10, b, i, j) \cup \exists_{b:\mathbb{B}, i, j:Z} env(900, b, i, j)]} Y (BDo_1, RDo_1, HS_1))) \quad (10) \end{aligned}$$

This is a fairly complicated formula, for which the explanation is as follows:

The νX fixed point, starts with an infinite search while keeping track of the Boiler asking for heat (*BDo*), the Radiator asking for heat (*RDo*) and whether the heat is served for the Boiler (*HS*).

If action *B1.DoWarmUp()* is encountered, variable *BDo* is set to *true* and if action *B1.DontWarmUp()* is encountered, it is set to *false*. Similar for *R1*.

When action *env(10, b, i, j)* is encountered, representing the values of the environment variables. The values of the second, third and fourth parameter are not of interest, therefore

the existential quantifier is used over these data parameters. The first value represents the Boiler temperature in tenths of a degree Celsius. When this is ten, the heat need of the Boiler is not satisfied, thus variable HS is set to *false*. If the similar action env , but then with the first value 900 is encountered, the heat needed is met and the variable HS is set to *true*.

If the action $HU1_B1.BurnerOn()$ is reached, it is checked that the Boiler and the Radiator do not need heat, because then searching further is not necessary, but if the Boiler and/or the Radiator needs heat, a new search infinite (νY) search is started, where again it is recorded whether the Boiler or Radiator is asking for heat and whether the heat for the Boiler is met.

If the variable BDo_1 and HS_1 are both *true* the formula continues with a finite search for a $B1.DontWarmUp()$ action, or, to break the finite path, a $R1.DoWarmUp()$, $B1.DoWarmUp()$ or an $env(10, b, i, j)$ for arbitrary boolean values for b and integer values for i and j .

If the variable BDo_1 is *false* and RDo_1 is *true* the formula continues with a finite search for a $R1.DontWarmUp()$ action, or to break the finite path, a $R1.DoWarmUp()$, $B1.DoWarmUp()$ or an $env(10, b, i, j)$ for arbitrary boolean values for b and integer values for i and j .

When the action $B1.DoWarmUp()$ is encountered it starts the second infinite search again with variable BDo_1 set to *true*. For the Radiator similar.

Also in the second search the variable HS_1 is updated according to the environment actions encountered.

If other actions than the above occur the infinite search is continues with unchanged variables.

Checking this formula yields as result *true*.

Property 4: Burner Off

After the Burner $HU1_B1$ is switched on and before the Burner is switched off, the Boiler and the Radiator do not need heat anymore.

$$\begin{aligned} & [true^* \cdot HU1_B1.On() \cdot \overline{B1.DontWarmUp() \cup R1.DontWarmUp()}^* \cdot \\ & HU1_B1.Off()]false \end{aligned} \tag{11}$$

Formula 11, stating that ‘between a Burner On and a Burner Off action the Boiler or the Radiator is warmed up and do not need heat anymore’, was checked with the following command:

```
$ lps2pbes -f31.mcf model.lps | pbes2bool -rjittyc -s3 -c
```

The result returns *false*, with a counterexample. The counterexample path has a Boiler $DoWarmUp$ action, after which the Burner is turned On. The Boiler is then warmed up, so the Boiler $DontWarmUp$ action is performed. Hereafter the Burner is turned On again and then turned Off. So between the second turning On of the Burner and the turning Off of the Burner no $DontWarmUp$ action is performed, hence the property is violated.

The cause for this property to fail is that the Burner acts upon a value that is set in the round before, making it possible to turn the Burner On while it was already On. This however is not harmful, since nothing changes in the state of the Burner.

A formula that takes the turning on of the Burner, possibly multiple times, into account

and the warming up and not warming up of the Boiler and Radiator also, is the following:

$$\begin{aligned}
& \nu X(BO : \mathbb{B} = false, BA : \mathbb{B} = false, RA : \mathbb{B} = false). \\
& \quad [HU1_B1.On()]X(true, BA, RA) \wedge \\
& \quad [HU1_B1.Off()]X(false, BA, RA) \wedge \\
& \quad [B1.DoWarmUp()]X(BO, true, RA) \wedge \\
& \quad [B1.DontWarmUp()]X(BO, false, RA) \wedge \\
& \quad [R1.DoWarmUp()]X(BO, BA, true) \wedge \\
& \quad [R1.DontWarmUp()]X(BO, BA, false) \wedge \\
& \quad \overline{[HU1_B1.On() \cup HU1_B1.Off() \cup B1.DoWarmUp() \cup B1.DontWarmUp() \cup} \\
& \quad \quad \overline{R1.DoWarmUp() \cup R1.DontWarmUp()]}X(BO, BA, RA) \wedge \\
& \quad [HU1_B1.On()]\nu Y(BA_1 : \mathbb{B} = BA, RA_1 : \mathbb{B} = RA).\mu Z.(\\
& \quad ((\neg BA_1 \wedge \neg RA_1) \Rightarrow \overline{[HU1_B1.Off() \cup B1.DoWarmUp() \cup R1.DoWarmUp()]}Z \wedge \\
& \quad \quad \langle true \rangle true) \wedge \\
& \quad [B1.DoWarmUp()]Y(true, RA_1) \wedge \\
& \quad [R1.DoWarmUp()]Y(BA_1, true) \wedge \\
& \quad [B1.DontWarmUp()]Y(false, RA_1) \wedge \\
& \quad [R1.DontWarmUp()]Y(BA_1, false) \wedge \\
& \quad \overline{[B1.DoWarmUp() \cup B1.DontWarmUp() \cup R1.DoWarmUp() \cup} \\
& \quad \quad \overline{R1.DontWarmUp() \cup HU1_B1.Off()]}Y(BA_1, RA_1)) \tag{12}
\end{aligned}$$

In Formula 12 a the variable BO represents whether the Burner is on, BA whether the Boiler is asking for heat and RA whether the Radiator is asking for heat. The corresponding actions that influence this are listed below the first line. If a different action was performed, the variables are not updated. If the Burner is set on, also an infinite search is started, where only the variables BA_1 and RA_1 , which have the same meaning as the BA and RA variables, respectively. If the Boiler and Radiator are not asking for heat, then a finite search is started for the Burner Off action. The search can be broken, when a Boiler DoWarmUp or Radiator DoWarmUp action is performed. The Do- and DontWarmUp actions update the parameters of the νY fixed point. If another action than the actions mentioned before, the variables are not updated.

Results

Verifying this formula yields *true*. The transitions and states at which the Burner Off action occurs, are the following:

1. the state INITIALIZING in State Machine of the HeatController
2. the transition from state ADJUST_BURNER to state CONTROL_BURNER
3. the transition from state CONTROL_BURNER to state WAIT_FOR_HEAT_REQUEST

Option (1) is performed only once, during the initialization phase. This action is not verified in the νY part, because no Burner On action has taken place before.

Option (2) is performed when the guard NOT(MDL_B1.bWarmUp OR MDL_R1.bWarmUp) is true, i.e., this corresponds exactly to the part in the formula $(\neg BA_1 \wedge \neg RA_1)$ that guards the finite search for the Burner Off action. It is possible that this transition becomes true when there was no Burner On action performed prior to that, so then it is also not taken into account.

Option (3) is performed when the guard NOT(MDL_HU1_B1.Q_BurnerSetpoint > 0.0) is true. This means that the Burner setpoint is set to 0. This is done when the values of the previous Burner setpoint and the Maximum percentage of heat needed are as shown in Table 6. This maximum percentage is obtained, from the percentages of heat needed in the Boiler and the Radiator.

Property 5: Burner vs Pump

When asked to warm up, the Burner $HU1_B1$ and the RegularPump $P1$ must go On.

previous Burner setpoint	Maximum percentage heat needed
5	< -5
4	< -6
3	< -7
2	< -8
1	< -9
0	< 11

Table 6: Previous Burner setpoints and Maximum percentages heat needed to make the new Burner setpoint 0

The formulas that are used are the following:

$$[true^* \cdot B1.DoWarmUp()] \mu X. [\overline{HU1_B1.On()}] X \wedge \langle true \rangle true \quad (13)$$

$$[true^* \cdot B1.DoWarmUp()] \mu X. [\overline{P1.On()}] X \wedge \langle true \rangle true \quad (14)$$

$$[true^* \cdot R1.DoWarmUp()] \mu X. [\overline{HU1_B1.On()}] X \wedge \langle true \rangle true \quad (15)$$

$$[true^* \cdot R1.DoWarmUp()] \mu X. [\overline{P1.On()}] X \wedge \langle true \rangle true \quad (16)$$

Results

Formula 13, stating that after the Boiler needs to be warmed up, inevitably the Burner must go on, was checked with the command:

```
$ lps2pbcs -f19.mcf model.lps | pbcs2bool -s3 -c -rjittyc
```

The result is *false*, because the Burner was already on when at a certain moment the Boiler needed to be warmed up. After the Boiler has been warmed up, the Burner is turned off and not turned on, because there is no heat demanded.

To incorporate that the Burner can be already on, the following formula is used:

$$\begin{aligned} & \nu X (bBOn : \mathbb{B} = false). [HU1_B1.On()] X (true) \wedge \\ & [HU1_B1.Off()] X (false) \wedge \\ & \overline{[HU1_B1.On() \cup HU1_B1.Off()]} X (bBOn) \wedge \\ & [B1.DoWarmUp()] (bBOn \vee (\mu Y. [\overline{HU1_B1.On()}] Y \wedge \langle true \rangle true)) \end{aligned} \quad (17)$$

Verification yields *true*.

Formula 14, stating that after the Boiler needs to be warmed up, inevitably the Pump must go on, was checked with the command:

```
$ lps2pbcs -f24.mcf model.lps | pbcs2bool -s3 -c -rjittyc
```

The result is *false*, because the Pump was already on when at a certain moment the Boiler needed to be warmed up. After the Boiler has been warmed up, the Pump is turned off and not turned on, because there is no heat demanded.

To incorporate that the Pump can already be on, the following formula is verified:

$$\begin{aligned} & \nu X (bPOn : \mathbb{B} = false). [P1.On()] X (true) \wedge \\ & [P1.Off()] X (false) \wedge \\ & \overline{[(P1.On() \cup P1.Off())]} X (bPOn) \wedge \\ & [B1.DoWarmUp()] (bPOn \vee (\mu Y. [\overline{P1.On()}] Y \wedge \langle true \rangle true)) \end{aligned} \quad (18)$$

The verification yields *false* again. This is caused by the fact that the Temperature of the Boiler is increased when only the Burner is On and the ThreeWayValve opened to the left and thus the BoilerController executes the Boiler DontWarmUp action and after that the Burner is turned off by the HeatController. Probably this fault was introduced, due to the fact that some timers are very short, but when the Timers are set to their true values, the resulting mCRL2 specification describes a process that is too large to verify. Formula 15, the Radiator equivalent of Formula 13 does yield *true*

Formula 16, stating that after the Radiator needs to be warmed up, inevitably the Pump must go on, was checked with the command:

`$ lps2pbcs -f1.mcf model.lps | pbcs2bool -s3 -c -rjittyc`

This returned as result *false* and a counterexample. The same cause as with Formula 14 was observed here. A similar formula as Formula 18 was created and verified, and there also the same result did appear.

5.1.2 Safety properties

The properties in this section, state that something bad must not happen.

Property 6: Controlled Valve Open and Burner On

ControlledValve HU1_CV2 must be opened before Burner HU1_B1 is switched on.

$$\overline{[HU1_CV2.Open()^* \cdot HU1_B1.On()]}false \quad (19)$$

$$[true^* \cdot HU1_CV2.Close() \cdot \overline{HU1_CV2.Open()}^* \cdot HU1_B1.On()]}false \quad (20)$$

The Parameterized Boolean Equation System (PBES) obtained from `lps2pbcs` was solved with `pbcs2bool`, but this takes a very long time. Therefore abstraction with `pbcsabstract` is used. The abstraction is done from all actions, except the actions that influence the actions of the formula. These are for Formula 19: the variables regarding the state of the HeatController state machine and MDL_HU1_Q_BurnerSetpoint_value_Var, because it is used on the guard before the actions. For Formula 20 these are the same as for Formula 19, but also MDL_B1_bWarmUp_value_Var and MDL_R1_bWarmUp_value_Var are not abstracted from.

Result

For both formulas, the tool gives with the ‘true’-abstraction the result *true*, which says nothing about the original formula, however the ‘false’-abstraction has also as result *true*, which means that the original system (and thus) formula returns *true*. When the `HU1_CV2.Open()` and `HU1_B1.On()` actions are looked up in the state machines that play a role in the system, they appear both only once and on the same transition in the correct order (Figure 10 on Page 25). So it is good that *true* is the outcome.

However Formula 19 gives true when it finds the first occurrence of a Burner On action which is preceded by a ControlledValve Open action. Interesting behavior, i.e., what happens after a ControlledValve Close action, is captured in Formula 20. To incorporate both in one formula, the following is used:

$$\begin{aligned} & \nu X(CV2 : \mathbb{B} = false).[HU1_CV2.Open()]X(true) \wedge \\ & [HU1_CV2.Close()]X(false) \wedge \\ & \overline{[HU1_CV2.Open() \cup HU1_CV2.Close()]}X(CV2) \wedge \\ & [HU1_B1.On()]CV2 \end{aligned} \quad (21)$$

The formula starts with an infinite search keeping track of the openness of ControlledValve HU1_CV2. Whenever a Burner On action is encountered, the variable CV2 must be true, meaning that the ControlledValve is open. Verification yields *true*.

Property 7: RegularPump On and Burner On

RegularPump P1 must be on before Burner HU1_B1 is switched on.

$$\overline{[P1.On()^* \cdot HU1_B1.On()]}false \quad (22)$$

$$[true^* \cdot P1.Off() \cdot \overline{P1.On()}^* \cdot HU1_B1.On()]}false \quad (23)$$

The same strategy as was applied to Property 6 is applied here. The actions that are not abstracted from are: MDL_HU1_Q_BurnerSetpoint_value_Var, MDL_HU1_B1_bFlameDetected, MDL_HU1_B1_lFlameDetected and the state machine state variables of the HeatController, PumpController, Burner and Environment. For Formula 23 also MDL_P1_iElapsedTime is of influence.

Result

In contrast with Property 6, where the solution was obtained as desired, this is not the case

here. Abstraction could be done from less variables, yielding a longer search. But with the intuition that this property really does not hold, a manual search for a counter example was started. With `lps2lts` a search for action `HU1_B1.On()` was performed. For Formula 22 a trace was found that did not contain a `P1.On()` action, hence contradicting the statement. And for Formula 23 a trace was found where a `P1.Off()` was present and after that no `P1.On()` action did occur, thus also contradicting this statement.

In the original system, this behavior can be explained as follows: The action `P1.On()` is guarded by `MDL_HU1_B1_bFlameDetected`. This variable denotes whether there is a flame detected in the Burner. This flame can only be detected if the Burner is On. Thus the Burner has to go On before the Pump goes On, thus contradicting the statement that the Pump must be on before the Burner is switched on.

For Formula 22 the same holds as for Formula 19: it only captures the truth of the first occurrence of a Burner On action preceded by a RegularPump On action. Here Formula 23 captures more interesting behavior: what happens after a Pump Off action. These can both be fit into the formula:

$$\begin{aligned}
& \nu X(p1 : \mathbb{B} = false).[P1.On()]X(true) \wedge \\
& \quad [P1.Off()]X(false) \wedge \\
& \quad \overline{[P1.On() \cup P1.Off()]}X(p1) \wedge \\
& \quad [HU1_B1.On()]p1
\end{aligned} \tag{24}$$

The formula starts with an infinite search keeping track of the RegularPump `P1` being on. Whenever a Burner On action is encountered, the variable `P1` must be true, meaning that the RegularPump is on. Verification yields *false* and the explanation is as given above.

Property 8: Timer

The property is as follows: For each timer, StartTimer must come before StopTimer. Since StartTimer and StopTimer are actions that occur at four elements (Boiler, Radiator, Burner and Pump), this property is represented by the following four formulas.

$$\overline{[B1.StartTimer()^* \cdot B1.StopTimer()]}false \tag{25}$$

$$\overline{[R1.StartTimer()^* \cdot R1.StopTimer()]}false \tag{26}$$

$$\overline{[HU1_B1.StartTimer()^* \cdot HU1_B1.StopTimer()]}false \tag{27}$$

$$\overline{[P1.StartTimer()^* \cdot P1.StopTimer()]}false \tag{28}$$

For these formulas abstraction is applied to abstract from certain variables. This is done in the same way as in Property 6. For the Boiler the variables of influence are: `s3_Coordinator`, `active_id.Coordinator`, `s2_MDL_BoilerController_SM1`, `MDL_BoilerController_SM_eCurrentState_MDL_BoilerController_SM1`, `MDL_BoilerController_SM_bATransitionWasPerformed_MDL_BoilerController_SM1`, `s6_MDL_RadiatorController_SM1`, `MDL_B1_TS2_lrTemperature_value_Var`.

For the Radiator these are: `s3_Coordinator`, `active_id.Coordinator`, `MDL_TH1_lrRoomTemperature_value_Var`, `MDL_TH1_lrTemperatureSetpointLow_value_Var`, `MDL_TH1_lrTemperatureSetpointHigh_value_Var`, `s2_MDL_BoilerController_SM1`, `MDL_R1_iElapsedTime_value_Var`, `MDL_R1_lrStartTemperature_value_Var`

For the Burner these are not recorded, since the outcome was not achieved by abstraction, but with verification on a reduced model.

For the Pump these are: `s3_Coordinator`, `active_id.Coordinator`, `MDL_HU1_B1_bFlameDetected_value_Var`, `MDL_P1_bTimerRunning_value_Var`, `MDL_P1_iElapsedTime_value_Var`, `s5_MDL_PumpController_SM1`, `MDL_PumpController_SM_eCurrentState_MDL_PumpController_SM1`, `MDL_PumpController_SM_bATransitionWasPerformed_MDL_PumpController_SM1`, `s12_RegularPump_RegularPump`.

Results

With all formulas the ‘true’-abstraction results in *true* and the ‘false’-abstraction results in: *true* for Formula 25. This can be verified in the original state machine in Figure 9 on Page 24, where the StartTimer and StopTimer are part of the entry and exit actions of one state, and thus the StartTimer action is performed when the state is entered and the StopTimer action is performed when the state is left, and thus a StartTimer comes always before a

StopTimer.

true for Formula 26. This can be verified in the same way as with Formula 25, but now with the state machine in Figure 10 on Page 25.

true for Formula 27. This can be verified in the same way as with Formula 25, but now with the state machine in Figure 13 on Page 27.

true for Formula 28. This can be verified in the same way as with Formula 25, but now with the state machine in Figure 11 on Page 26.

The argument given in Property 6 and 7, regarding the incompleteness of the formulas does also apply here: Only the truth of the first occurrence of a StartTimer - StopTimer pair is recorded. To find them all, the following formula is used:

$$\begin{aligned}
& \nu X(TS : \mathbb{B} = false).[B1.StartTimer()]X(true) \wedge \\
& \quad [B1.StopTimer()]X(false) \wedge \\
& \quad \overline{[B1.StartTimer() \cup B1.StopTimer()]}X(TS) \wedge \\
& \quad [B1.StopTimer()]TS
\end{aligned} \tag{29}$$

The formula represents an infinite search where it is recorded whether the timer is started. Whenever a StopTimer is encountered, the value of variable *TS* must be true, meaning that the timer was started. Validating the formula yields *true*. For the Burner, Radiator and RegularPump a similar formula is used and the verification yields for all of them *true* as well.

Property 9: StartTimer

The property is: No two StartTimer actions on the same object may occur without a StopTimer in between.

As with Property 8, this is represented by four formulas:

$$[true^* \cdot B1.StartTimer() \cdot \overline{B1.StopTimer()}^* \cdot B1.StartTimer()]false \tag{30}$$

$$[true^* \cdot R1.StartTimer() \cdot \overline{R1.StopTimer()}^* \cdot R1.StartTimer()]false \tag{31}$$

$$[true^* \cdot HU1_B1.StartTimer() \cdot \overline{HU1_B1.StopTimer()}^* \cdot HU1_B1.StartTimer()]false \tag{32}$$

$$[true^* \cdot P1.StartTimer() \cdot \overline{P1.StopTimer()}^* \cdot P1.StartTimer()]false \tag{33}$$

The abstracted variables are the same as with Property 8.

Results

Also the results and justification are the same as with Property 8.

Property 10: StopTimer

This property states: No two StopTimer actions on the same object may occur without a StartTimer in between.

This is represented by four formulas as in Property 8 and 9.

$$[true^* \cdot B1.StopTimer() \cdot \overline{B1.StartTimer()}^* \cdot B1.StopTimer()]false \tag{34}$$

$$[true^* \cdot R1.StopTimer() \cdot \overline{R1.StartTimer()}^* \cdot R1.StopTimer()]false \tag{35}$$

$$[true^* \cdot HU1_B1.StopTimer() \cdot \overline{HU1_B1.StartTimer()}^* \cdot HU1_B1.StopTimer()]false \tag{36}$$

$$[true^* \cdot P1.StopTimer() \cdot \overline{P1.StartTimer()}^* \cdot P1.StopTimer()]false \tag{37}$$

The abstracted variables are the same as with Property 8 and 9.

Results

Also the results and justification are the same as with Property 8 and 9.

Property 11: Final

A deadlock may only occur immediately after a ‘Final’ action.

$$\overline{[Final^*]}(true)true \tag{38}$$

This formula is a variation of the standard absence of deadlock formula. The difference is that instead that there must be an action possible in every state, now there must be an action possible in every state that is not reached by taking a Final action as the last action. For this formula the whole state space has to be inspected. There are a few things to notice about this property:

- Abstraction does not help, because all variables in the specification are influential for reaching all states.
- This formula is basically looking for the absence of deadlocks, except for those coming right after a Final action. Therefore a deadlock search was started with `lps-reach`, but it did find no deadlocks. If no deadlocks are found, the formula holds right away. But if one would have been found, other methods for deadlock search have to be taken, such as a deadlock search with `lps2lts`, where traces are recorded and afterwards are inspected on the occurrence of ‘Final’ as the last taken action.

Results

Since the deadlock search with `lps-reach` did not find any deadlock, it did also not find any deadlocks that did not have ‘Final’ as its last action and thus the formula holds.

Property 12: Open Both

ThreeWayValve CV1 must not be opened to both sides at the same time.

$$[true^* \cdot CV1.OpenBoth()]false \quad (39)$$

This formula can be checked with a reachability check, because it is the same as $\neg \langle true^* \cdot CV1.OpenBoth() \rangle true$. Thus if the reachability check returns true, the statement is false and vice versa. The reachability check is done as follows:

```
$ lps-reach --regroup=gs --mcr12-readable-edge-labels
--action='actThreeWayValve_OpenBoth(MDL_CV1)' model.lps
```

where the `ltsmin` tool `lps-reach` is used with some options. For explanation see Property 1.

Result

The result is that it cannot find the action, so the statement is *true*. This is the result with the semantics as used in the PLC code, i.e., if more than one transition is enabled, the first one in the specification is taken, thus it is possible that a transition is never taken, although it is enabled. The semantics according to the UML standard does create a state space where the action is reachable, because it allows each transition that is enabled to happen.

The trace obtained in the second case (UML semantics) could be mapped back to the state machines, although it was not trivial. With the proper understanding of the system it could be led back to the fact that a transition was taken that was not present in the diagram. In Figure 12 on Page 26 the state machine is shown as seen by the user, but in Figure 18 the state machine is shown as it was translated. The difference lies in the presence of the ‘OpenBoth’ state and the transitions connected to it. The error was introduced by deleting state ‘OpenBoth’ from the diagram, but not from the model, just meaning that the ‘OpenBoth’ state was hidden.

In the first case (PLC semantics), where the transition from ‘OpenRight’ to ‘OpenBoth’ is not taken, this is due to the fact that when that transition is enabled, i.e., `MDL_B1.bWarmUp AND NOT MDL_R1.bWarmUp` is true, also the transition from ‘OpenRight’ to ‘OpenLeft’ is enabled, i.e., `MDL_B1.bWarmUp OR MDL_R1.bWarmUp` is true. Since the transitions are checked in order, the ‘OpenRight’ to ‘OpenLeft’ transition is taken first and thus neglecting the other transition.

Property 13: Burner Off

It is impossible to turn Burner `HU1_B1 On` two times, without switching Burner `HU1_B1 Off` in between.

$$[true^* \cdot HU1_B1.On() \cdot \overline{HU1_B1.Off()}^* \cdot HU1_B1.On()]false \quad (40)$$

Formula 40, stating that ‘between two Burner On actions a Burner Off action should take place’, was checked with the following command:

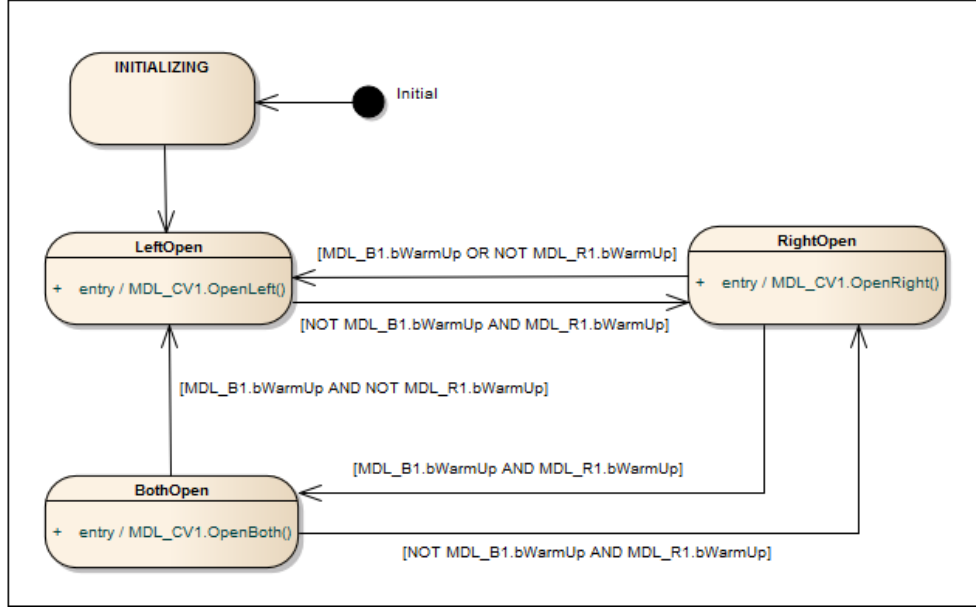


Figure 18: Real Three Way Valve Controller Model

```
$ lps2pbcs -f32.mcf model.lps | pbcs2bool -rjittyc -s3 -c
```

The result returns *false*, with a counterexample. The part of the example that has a ‘false’ in it is used to reconstruct the trace. This trace showed that indeed two Burner On actions could be done without a Burner Off action in between. The reason is that the Burner periodically checks whether there is still heat needed and if heat is needed, it calculates the percentage heat needed and then opens the Gas Valve and turns the Burner on again. Although it is strange that it happens, it is not a problem to open the Valve or turn the Burner on again, because internally nothing changes.

Property 14: Burner On

It is impossible to turn Burner *HU1_B1* Off two times, without switching Burner *HU1_B1* On in between.

$$[true^* \cdot HU1_B1.Off() \cdot \overline{HU1_B1.On()}^* \cdot HU1_B1.Off()]false \quad (41)$$

This formula was verified with the command:

```
$ lps2pbcs -f33.mcf model.lps | pbcs2bool -rjittyc s3
```

Results

The verification of Formula 41 yields *true*, when verified with a model where the Burner, Boiler and Radiator Timer checkpoints are set to 1. When verified with the timers set to 2, 10 and 10, respectively, the result yields *false*. This is caused by the fact that during the Boiler or Radiator Timer running, the Burner runs several times, with a Burner setpoint set to 0, but that causes the Burner Off action to be taken. The behavior is consistent with Property 13, and also in this case the multiple Burner Off actions, do not cause any harm.

Property 15: Controlled Valve

If Burner *HU1_B1* is not burning, the ControlledValve *HU1_CV2* must be closed.

This formula is ambiguous in the sense that it is not clear whether ControlledValve *HU1_CV2* must be closed before the Burner is turned off or after the Burner has been turned off. The first attempt of formulas representing the two possibilities are listed here:

$$\overline{[HU1_CV2.Close().HU1_B1.Off()]}false \quad (42)$$

$$\overline{[HU1_B1.Off().HU1_CV2.Close()]}false \quad (43)$$

Verification yields true for the first and *false* for the second. The counterexample of Formula 45 returns a very short trace to the first ControlledValve Close action, since it is not preceded

by a Burner Off action. As stated earlier in Properties 4, 5 and 8 these formulas do only say something about the first occurrence. To incorporate the other occurrences, the following formulas are used:

$$\begin{aligned} & \nu X(CV2 : \mathbb{B} = false).[HU1_CV2.Open()]X(true) \wedge \\ & \quad [HU1_CV2.Close()]X(false) \wedge \\ & \quad \overline{[HU1_CV2.Open() \cup HU1_CV2.Close()]}X(CV2) \wedge \\ & \quad [HU1_B1.Off()]\neg CV2 \end{aligned} \tag{44}$$

$$\begin{aligned} & \nu X(BO : \mathbb{B} = false).[HU1_B1.On()]X(true) \wedge \\ & \quad [HU1_B1.Off()]X(false) \wedge \\ & \quad \overline{[HU1_B1.On() \cup HU1_B1.Off()]}X(BO) \wedge \\ & \quad [HU1_CV2.Close()]\neg BO \vee \mu Y.\overline{[HU1_B1.Off()]}Y \wedge \langle true \rangle true \end{aligned} \tag{45}$$

Formula 44 represents an infinite search, where the openness of ControlledValve *HU1_CV2* is recorded. Whenever a Burner Off action takes place, the value of variable *CV2* must be false, meaning that the ControlledValve is closed. Verification yields *true*.

Formula 45 represents an infinite search, where it is recorded whether the Burner is on. Whenever a ControlledValve Close action is encountered, the value of variable *BO* must be false, meaning that the Burner is off. The verification of the formula as described, yields *false*, for the same reason as Formula 43 fails also. To make the formula *true*, the μ -part is added. This part searches for a finite path on which the Burner is turned Off. Verification of this formula yields indeed *true*.

Property 16: Max Temperature

The following requirements were not clear enough to come up with formulas, which capture the real intentions:

- Temperature of the Boiler B1 may never exceed the maximum allowed temperature.
- Temperature of the Radiator R1 may never exceed the maximum allowed temperature.
- Temperature of the water in the Central Heating System may never exceed the maximum allowed temperature.

Some questions that could be asked in relation to these requirements are the following:

- What is the maximum allowed temperature? Is that 'lrTemperatureSetPointHigh', the high setpoint of the Boiler or the Thermostat?
- What is the Central Heating System? Is that the Burner (Central Heating Unit) or the total of the pipes and elements? What is in the latter case the maximum temperature allowed?

Due to the domain expert, being ill, there could no answers be obtained from the domain expert.

5.2 SoLayTec Requirements

In this section the properties are described to which the SoLayTec system must adhere. Also the outcome of the verification or why it did not succeed to verify the property is explained.

5.2.1 No deadlock

The system should not end up in a deadlock state. This can be expressed by the modal *mu*-calculus formula:

$$[true^*]\langle true \rangle true \tag{46}$$

Due to the large state space resulting from linearizing the mCRL2 specification, this formula is not checked with `lps2lts` or `lps2pbcs` and `pbcs2boo1`. Instead symbolic reachability with `lps-reach` is used to explore the state space.

Result

In separate state machines some deadlocks were found, due to the reachability of some modeled error states.

5.2.2 Reachability

All states of the state machines should be reachable. For this purpose the names of the states are added as actions to the mCRL2 specification. To check this property, the formula

$$\langle true^* \cdot state_name \rangle true \quad (47)$$

is used, where *state_name* represents the state name for which the reachability check is done. The formulas are checked on the separate state machines in two types:

1. all guards are true.
2. the variables that influence the behavior are manipulated by the ‘environment’. This is done by setting the values of the variable to either their minimum or maximum value.

Results

For the first case in only one state machine some states could not be reached. The argument of SoLayTec was that this state machine was conceptual. The intended behavior was not modeled correctly.

In the second case, the same states could not be reached plus some other states too:

In one state machine one state (and a following state) was not reachable, because the guard on the transitions that lead to this state become never true. The guard checks whether a variable is at some warning level boundaries. This guard becomes never true, because the state machine and the environment, do never manipulate the value of the variable in such a way that it reaches these boundary values.

In another state machine a series of states is not reachable if only the environment and the state machine are modeled in detail. This was solved by adding the processes that also influence the environment.

Note that the outcome can be different if all state machines are put in parallel and data values block or enable certain transitions.

5.2.3 No Interlocks

The system should not have interlocks. Interlocks are states in the system that are not allowed. The interlocks specified by SoLayTec can be categorized into three property groups. Since all these properties deal with data, the most convenient way of specifying the properties is as conditionals in the variable process. The conditional guard, guards an action called error. As data parameter for this action a positive number is assigned. To verify this properties a simple reachability check for this error action is sufficient. This is again done with symbolic reachability analysis with `lps-reach`.

Due to limited time and the complexity of the model it was not feasible to get results for the described properties.

Property 1

The human readable form of the property:

1. Setpoint CoriFlow > '0' and Setpoint MFC101 < '1' should not be possible because of contamination risk N₂ main supply with TMAL (always).

This property states that the flow set point for the Liquid Flow Meter (LFM or CoriFlow) and Mass Flow Controller (MFC) elements must be below or above a certain value. If that is not the case, the system is in error. The mCRL2 code corresponding to this property:

- 1 (LFM101_MDLI_FlowSetPoint_value > 0 &&
- 2 MFC101_MDLI_FlowSetPoint_value < 1) -> error(1) . delta

Property 2

The general form of this property is concerned Pressure Valves. It prohibits that a certain valve is opened when another valve is already open. Here the first human readable property:

2. Opening PV114 if PV103 is open should not be possible because of contamination risk N₂ main supply with TMAL (always)

The mCRL2 formula is:

- 1 (PV103_MDLOU_IsOpen_value && PV114_bIsOpening_value) -> error(2) . delta

The second property is the converse of the previous:

3. Opening PV103 if PV114 is open should not be possible because of contamination risk N₂ main supply with TMAL (always)
In mCRL2 code:

- 1 (PV114_MDLOU_IsOpen_value && PV103_bIsOpening_value) -> error(3) . delta

The next formula and its converse are:

5. Opening PV301 if PV308 is open should not be possible because of shared orifice OR302 (only possible in manual mode SoLayTec service engineer) and

6. Opening PV308 if PV301 is open should not be possible because of shared orifice OR302 (only possible in manual mode SoLayTec service engineer)
which in mCRL2 are:

- 1 (PV301_MDLOU_IsOpen_value && PV308_bIsOpening_value) -> error(5) . delta
- 2 +
- 3 (PV308_MDLOU_IsOpen_value && PV301_bIsOpening_value) -> error(6) . delta

Idem for property 11 and 12:

11. Opening PV107 if PV108 is open should not be possible. Avoid gas cabinet abatement line and process line to be open simultaneously (only possible in manual mode SoLayTec service engineer) and

12. Opening PV108 if PV107 is open should not be possible. Avoid gas cabinet abatement line and process line to be open simultaneously (only possible in manual mode SoLayTec service engineer)
which in mCRL2 are:

- 1 (PV108_MDLOU_IsOpen_value && PV107_bIsOpening_value) -> error(11) . delta
- 2 +
- 3 (PV107_MDLOU_IsOpen_value && PV108_bIsOpening_value) -> error(12) . delta

Property 3

This property looks like Property 2, but instead of the condition that a Pressure Valve is open, now the pressure values of two Pressure Transmitters are compared. If this is in the wrong position in combination with the opening of a Pressure Valve, the system is in error.

4. Opening PV114 if PT102 > PT205 should not be possible because of contamination risk N₂ main supply with TMAL (only possible in manual mode SoLayTec service engineer)

- 1 (PT102_lrPressure_value > PT205_lrPressure_value &&
- 2 PV114_bIsOpening_value) -> error(4) . delta

The next properties do also take an offset into account while comparing the pressure values.

7. Opening PV307 if PT305 (minus offset) > PT301 should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

- 1 (PT305_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT301_lrPressure_value &&
- 2 PV307_bIsOpening_value) -> error(7) . delta

8. Opening PV305 and PV306 (in which order and which time interval doesn't matter) if PT305 (minus offset) > PT302 should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

- 1 (PT305_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT302_lrPressure_value &&
- 2 PV306_bIsOpening_value) -> error(8) . delta

9. Opening PV301 if $PT301 \text{ (minus offset)} > PT303$ should not be possible because of contamination risk N_2 main supply with TMAL (only possible in manual mode SoLayTec service engineer)

1 (PT301_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT303_lrPressure_value &&
2 PV301_bIsOpening_value) -> error(9) . delta

10. Opening PV313 if $PT301 \text{ (minus offset)} > PT303$ should not be possible because of contamination risk N_2 main supply with TMAL (only possible in manual mode SoLayTec service engineer)

1 (PT301_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT303_lrPressure_value &&
2 MDL_PV313_bIsOpening_value) -> error(10) . delta

13. Opening PV103 if $PT101 \text{ (minus offset)} > PT205$ should not be possible to prevent a higher pressure on the liquid side then on the N_2 side of the CEM and the possible contamination risk N_2 main supply with TMAL (only possible in manual mode SoLayTec service engineer)

1 (PT101_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT205_lrPressure_value &&
2 PV103_bIsOpening_value) -> error(14) . delta

14. Opening PV103 if $PT102 \text{ (minus offset)} > PT101$ should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

1 (PT102_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT101_lrPressure_value &&
2 PV103_bIsOpening_value) -> error(15) . delta

15. Opening PV112 if $PT104 \text{ (minus offset)} > PT205$ should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

1 (PT104_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT205_lrPressure_value &&
2 PV112_bIsOpening_value) -> error(16) . delta

16. Opening PV108 if $PT104 \text{ (minus offset)} > PT103$ should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

1 (PT104_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT103_lrPressure_value &&
2 PV108_bIsOpening_value) -> error(17) . delta

17. Opening PV317 if $PT305 \text{ (minus offset)} > PT303$ should not be possible because of back flush risk (only possible in manual mode SoLayTec service engineer)

1 (PT305_lrPressure_value - OFFSET_PRESSURE_SMALL) > PT303_lrPressure_value &&
2 PV317_bIsOpening_value) -> error(18) . delta

6 Guidelines for Modeling with Formal Verification in Mind

In this section the guidelines as recorded during the development process are listed. They are split into modeling guidelines (Section 6.1) and verification guidelines (Section 6.2). The guidelines given should be taken into account when developing models, which should also be suitable for formal verification in the way as is described in this thesis.

At a high level the guidelines are as follows:

1. For Formal verification, the model must be sufficiently precise to have the needed information for verification and sufficiently abstract such that verification is doable. For instance hand written code that is of importance must be incorporated in the model in some way.
2. The language used in guards, effects and triggers, must have a clear syntax and semantics. In this thesis, the PLC code language is used, but when the output of the code generation process is not PLC code, the guards, effects and triggers most likely contain different syntax and semantics. To make model checking generation and also modeling in general more robust, a language must be used, that is both concise and expressive enough to model the artifacts that correspond to the target language. This has advantage in modeling, because there is a common language to reason with. It has also an advantage in model checking, since a parser has to be made only once, which can be used in the translation to mCRL2 and any other desired language. If the PLC language is chosen, the guidelines are explained in more detail in Section 6.1.
3. In model checking different tasks are discerned, such as reachability or deadlock detection. For each task a certain recipe must be made and followed. Some of these recipes are documented in Section 6.2.

6.1 Guidelines for modeling

For the generation of a good specification for the use of model checking, there are some guidelines to follow. They are clustered in guidelines for the Framework accompanying the Domain-specific language profile and in guidelines for the model, made with this language.

Framework

- The meta model (profile) file has to contain a ‘Framework’ package. This package must contain a class diagram in which the classes represent the abstract elements represented by the meta model.
This description of the variables, constants and actions that belong to the element is used by the generator to make a specification that is complete enough to model the behavior in enough detail to make model checking with data possible.
- The attributes of the element class should represent the variables corresponding to that element as used in the target language of the code generation (in this document: a PLC function block). The attributes must be statically typed:

int for all integer types

double for all real types

boolean for all boolean types

For the generation of the mCRL2 specification with variables and constants, the type information must be part of the model.

- The attributes must have a default value (initial value) and for the integer and real values, also an upper and lower bound must be provided.
The default value is necessary to instantiate an element with an initial value. Since integer and real values have an infinite domain, the mCRL2 specification (or any other code) would in general be infinite as well. To help prevent this, the domain is bounded. Note that for real values the domain is still infinite, therefore the real variables must directly or indirectly be dependent on integer or constant values.

- If the attribute must represent a constant, it should be marked as such and then an upper and lower bound are not needed.
This is done to distinguish between variables and constants. The upper and lower bound are not needed, because for a constant the upper and lower bounds are the same as the value of the constant. Besides that, the representation of a constant in the mCRL2 specification is done by a function taking zero arguments and returning the value.
- The operations of the element class should represent the actions and behavior of that element or the manipulations of the attributes of that element. Since operations represent behavior, use state machines to model this. The state machine has the same name as the operation to link these two in the generation process.
- The general behavior of an element, must be represented by an operation and a corresponding state machine.
This is done, because the modeled behavior can influence the state of the system and is thus of importance for the mCRL2 specification.
- The state machine modeling this kind of behavioral actions must consist of at least an Initial state and a Final state with a transition path from the Initial state to the Final state. On this path one or more simple states are allowed. The transitions may contain a guard and/or a list of assignments (effect). A list of assignments must not end with a semicolon (;).
These restrictions are made, because then the generator and parser can handle them correctly.
- The hand written `IF guard THEN effect ENDIF` constructs in PLC code must be translated into the state machine construct in Figure 19. The dotted arrows denote connections to states before and after these states. If no connected states are before or after this construct, State1 is an Initial state and State2 is a Final state.
The translation is done in this way, so at least one path is followed, because otherwise there would be a deadlock when the guard is false.

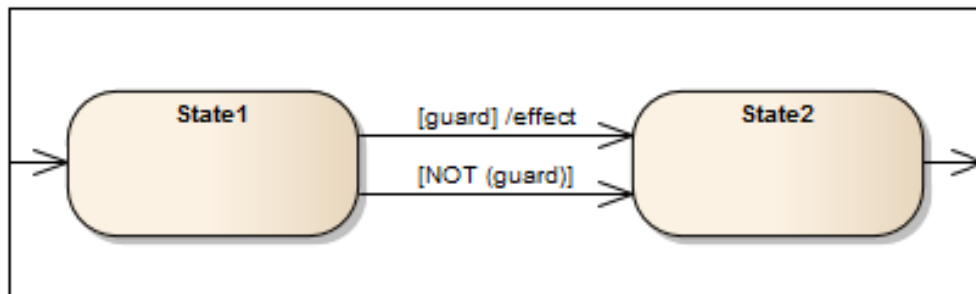


Figure 19: If-then-else construct in State Machine format

Model

- A Controller element in the domain-specific model must contain one or more state machines that control the elements in the model and the Controller must be active.
If modeled in this way, the generator will translate the state machine, otherwise it will not.
- Guards and assignments can contain global variables and constants. These should be added as attributes to the Controller element the state machine is part of. The same recommendations as above according to types, default values, bound values and constants apply.
This is again done, to model the behavior of the system in such detail as is necessary to check the properties in a decent way.
- Global actions must be modeled with operations as part of the Controller element and must have a corresponding state machine.

- Action calls as part of effects must be postfixed with ‘()’.
This must be done to distinguish actions from variables.
- To avoid the state space explosion problem, there should be only a few state machines, which have only a few states. Most important is to use as less data as possible and otherwise, they should have small finite domains, such as the booleans.

6.2 Guidelines for model checking

Since the model checking is not done automatically, but with manual use of the mCRL2 toolset, some guidelines are given to work with the toolset in a productive way.

6.2.1 Properties

The properties to which the modeled system must adhere can be specified in several ways. Here follow some guidelines:

- For *reachability*, the standard form of the μ -calculus formula is:

$$\langle true^* \cdot action \rangle true$$

where *action* is the action (with its parameters) for which the reachability is checked. There are several ways to get a result with a linear process specification (LPS) obtained from linearizing the generated mCRL2 specification:

- Use `lps2lts` with the flags `--action=action` for reachability check of the specified action (or comma separated list of actions) and `--trace` for trace recording. Note that here *action* must be used without parameters.
 - Use `lps2pbes` with the flag `--formula=<file>` to generate a parameterized boolean equation system, where `<file>` is a `*.mcf` file containing a μ -calculus formula in machine readable format. In this case `<true*.action>true`, where *action* is the action for which the reachability is checked. The obtained parameterized boolean equation system is then solved by `pbes2bool` with flags `--strategy=2` or `3` and `--counter` to generate a counter example if the outcome is false and returns a witness if the outcome is true.
 - Use `lps-reach` with the options `--regroup=gs` (where *gs* stands for Group Safely), `--mcr12-readable-edge-labels` and `--action=action` to use a tool that performs a symbolic reachability check for the provided action.
- For *deadlock*, the standard form of the μ -calculus formula is:

$$[true^*] \langle true \rangle true$$

If for example only after a *Final* transition a deadlock is allowed, the first *true* can be replaced by *Final*. Also here are several ways to check for deadlock. Note that these ways work all for the standard form.

- Use `lps2lts` with the flags `--deadlock` for deadlock detection and `--trace` for trace recording of the trace to the deadlock. For the alternative form, every trace may only have the *Final* action as its last action.
 - Use `lps2pbes` in the same way as with reachability. In this case the formula file contains the μ -calculus in the standard or the alternative form.
 - Use `lps-reach` with the flags `--regroup=gs` and `--deadlock` to let the tool perform a search for a deadlock. Note that this tool terminates after the first deadlock has been found.
- For more complex properties which involve actions, it is best to formulate a μ -calculus formula. Note that this is most of the time not very intuitive or easy to come up with a formula that expresses what you mean. The tools to use are `lps2pbes` and `pbes2bool` as described in the tools that can be used for a reachability check. A few patterns that did occur in this thesis are given here:
 - The pattern: ‘Between two actions *actionA* an action *actionB* must occur’. The μ -calculus formula for this pattern is:

$$[true^* \cdot actionA() \cdot \overline{actionB()}^* \cdot actionA()] false$$

It represents that in every reachable state, where an *actionA* action can be done, it is not possible to reach an *actionA* action again, without an *actionB*

- The pattern: ‘If action $actionA$ happens, then action $actionB$ must happen’. With this pattern it is often the case that action $actionB$ may also be done before action $actionA$, i.e., action $actionB$ influences the state S of the system. The μ -calculus formula is the following:

$$\begin{aligned} & \nu X(S_{status} : \mathbb{B} = false). \\ & \quad [A_{Bad}]X(false) \wedge \\ & \quad [A_{Good}]X(true) \wedge \\ & \quad \overline{[A_{Bad} \cup A_{Good}]}X(S_{status}) \wedge \\ & \quad [actionA()]S_{status} \vee \mu Y.\overline{[A_{Good}]}Y \wedge \langle true \rangle true \end{aligned}$$

where A_{Bad} and A_{Good} are the sets of actions that influence the state S negatively and positively. Action $actionB()$ is part of set A_{Good} . The inverse of the union of these sets ($\overline{[A_{Bad} \cup A_{Good}]}$) is the set of actions that do not influence state S . The action $actionA$ is the triggering action in the pattern. The $S_{status} \vee \mu Y.\overline{[A_{Good}]}Y \wedge \langle true \rangle true$ part represents the second part of the pattern, i.e., the state is already good (S_{status}), or no loop or finite path exists on which an action in A_{Good} does not occur.

- The pattern: ‘If action $actionA$ happens, then action $actionB$ must happen or action $actionC$ may happen unboundedly often, before that. The μ -calculus formula is as follows:

$$\begin{aligned} & \nu X(S_{status} : \mathbb{B} = false). \\ & \quad [A_{Bad}]X(false) \wedge \\ & \quad [A_{Good}]X(true) \wedge \\ & \quad \overline{[A_{Bad} \cup A_{Good}]}X(S_{status}) \wedge \\ & \quad [actionA()]S_{status} \vee (\nu Y(S1_{status} : \mathbb{B} = S_{status}). \\ & \quad \quad \mu Z.(\neg S1_{status} \Rightarrow \overline{[A_{Good} \cup A_{Break}]}Z \wedge \langle true \rangle true) \wedge \\ & \quad \quad [A_{Break}]Y(false) \wedge \\ & \quad \quad [A_{Unbreak}]Y(true) \wedge \\ & \quad \quad \overline{[A_{Break} \cup A_{Unbreak} \cup A_{Good}]}Y(S1_{status})) \end{aligned}$$

where the $A_{..}$ sets have the same meaning as in the previous pattern. The A_{Break} set contains actions that break the finite search (the μZ -search), for example, because that action starts an infinite path (νY -search) on which the searched for (A_{Good}) actions do not occur. Whenever this infinite path (or loop) is exited by an action in the set $A_{Unbreak}$, the finite search is started again.

Note that the state of the system is here recorded in one variable. It is possible to model more complex states with more variables or with variables of a different type.

- For properties where only data is involved, it is better to model these properties as guards, followed by an *error* action or another action which expresses the result of the guard being true. The construct you get is (guard) \rightarrow error . delta in an mCRL2 process that has the variables that occur in the guard as parameters. The delta process is to make the action error end in a deadlock state. This is done to help create more ways to search for the error:
 - Do a reachability check on the *error* action.
 - Do a deadlock check and inspect the trace or counter example to verify that the deadlock found is one caused by the *error* action.

6.2.2 Tools

Some tools can be substituted by others. Instead of `lps2pbes`, `lts2pbes` can be used with a Labeled Transition System (LTS) obtained by `lps2lts` and possible manipulations of the LTS and instead of `pbes2bool`, `pbespgsolve` can be used, to solve the Parameterized Boolean Equation System with an other algorithm. In general `lps-reach` is the fastest tool for reachability check.

The Linear Process Specifications and Parameterized Boolean Equation Systems (PBES) can be optimized by using various manipulation tools. See Appendix C.1 for a description of these tools. One tool is mentioned here: `pbabstract`. With this tool some variables of the PBES are set to *true* or *false* (called abstraction) to approximate the solution to the system:

- If variables are set to *true*, the solution is made more true, so if then the solution returns *true*, there cannot be said anything about the original formula, but if it returns *false* the original formula was also *false*.
- If the variables are set to *false*, the solution is made less true, so if then the solution returns *true*, the original formula was also *true*, otherwise, nothing can be said about the original formula.

Note that the variables that are left out, must not influence the property. By abstracting from the correct variables this tool can improve the solving time of the PBES immensely. A downside of the program is that in some cases, there cannot be said anything about the original formula, and several tries may be needed to select the right variables. Another downside is that if a counter example is provided, it can be hard to reconstruct the trace that caused it, because some relevant trace information is abstracted away.

A performance difference can also be obtained to use the compiling rewriter, with the flag `--rewriter=jittyc` instead of the normal rewriter. The rewriter is a part of many tools. Note that the compiling rewriter is not available on the Windows platform.

7 Related Work

State charts, similar to state machines, were formalized by David Harel [15] in the eighties of the 20th century. It was intended to be a visual formalism to represent complex systems. After his proposal, several other people have proposed different formalisms, but also different semantics. It has been clear that it is hard to say what the ‘official’ semantics are. The Object Modeling Group [23] nowadays provide the standard for the state machines as part of the Unified Modeling Language (UML).

In [25] a formalization is given for the UML state machine semantics as seen by the writers. The version of the standard is there 1.3 and the version in this thesis 2+ is used. Their formalization is used in model checking as well. In their vUML tool [21] they use the PROMELA language, the input language for SPIN, for model checking. The translation of UML to PROMELA in vUML makes use of class diagrams, collaboration diagrams and state machine diagrams.

In [20] the approach is almost the same as used in this document. The approach there is that the Programmable Logic Controller the basis is and modeled by UML state charts. The UML state charts are then translated to the formal verification language nuSMV.

State Machines are closely related to finite state machines. In [30] the finite state machines as captured in Stateflow diagrams as part of Simulink are translated to mCRL2 for model checking. The INESS Project [14] uses a similar approach, where there is made a model transformation from Executable UML Models (xUML) to mCRL2 processes.

Another way of model checking is described in [16]. It is based on Live Sequence Charts, instead of State Charts. Their approach was worked out in Java. An advantage of their approach is that they can leave the transformation to a verification language out.

In [4] an overview of model checking of statechart models is given, where different approaches are discussed. Part of these approaches are based on SPIN with PROMELA, others on SMV or UPPAAL [2]. All have different requirement specification languages that can be used.

It is well known that it is always hard to formulate requirements or properties of a system in a non-ambiguous way. An approach taken in [26] is to use specification patterns and parse / translate these to formal requirements. A drawback would be that some expressiveness can be lost, especially with a rich language as the *mu*-calculus.

In some tools for model checking the results are given in a trace, for example in vUML [21] and UPPAAL. These traces are often fed back to the user by means of a Message Sequence Charts [10, 19]. In such a chart the elements are listed horizontally as a box with a vertical lifeline and the actions (messages) are visualized vertically, as horizontal arrows between the lifelines.

8 Conclusions and Future Work

In this section the results and conclusions are given as a response to the problem put forward in the introduction. Since this report handles the first steps towards successful model checking also future work is mentioned.

8.1 Problem Revisited

The problem investigated in this thesis is to see whether mCRL2, as a means for model checking, is useful in the context of Sioux to add model checking and analysis to model driven software development in a multi-disciplinary setting. The systems and models that need to be developed are getting more and more complex, partly due to the multi-disciplinary setting. This demands for a way of verification of the models, to be sure that the complex system adheres to the intended specification.

8.2 Results/Conclusions

To get an answer to the problem, an infrastructure was developed to make model checking possible. This consists of

1. guidelines for modeling and model checking and
2. a generator that can take the (meta)models and generate an mCRL2 Specification.

These (meta)models involve a domain specific language to describe the structure of the modeled system and the state machine language to describe the behavior of the (elements of the) modeled system.

The properties to which the system should adhere were provided by the domain expert. The human readable properties were manually transformed into formal properties using the μ -calculus or the mCRL2 language. With these formal properties model checking was applied to the system. To get solutions to the properties, several tools of the mCRL2 toolset and LTSmin toolset were used.

While this approach was applied to two different cases, several flaws of the modeling could be shown.

In the Language Workbench Challenge 2012 Case there were several properties that could be proven true, and counter examples were given, for the properties that were not valid. For the SoLayTec Case the reachability checks could be performed.

When the work done on the SoLayTec case was shown at SoLayTec, the only result at that time was that there were some states not reachable. Although they see the potential in model checking, the current drawback is the usability. The generation of the mCRL2 specification can be done mostly automatically, but the model checking is all done by manually using the tools of the toolsets. To use these effectively one needs to have quite some expert knowledge and insight into mCRL2 and μ -calculus. The ideal situation for them would be that they provide the (meta)models and with one push of a button they get the results: either everything is okay or in an understandable way feedback where possible flaws are found.

The conclusion is that model checking as a method to improve software quality in an early stage will add value to projects done at Sioux, but one needs an expert to facilitate this, and / or more effort should be put in making model checking more usable by the domain expert. The guidelines can be used in future projects concerning model checking.

8.3 Future Work

Model checking gives added value, but only when usable. For this usability and applicability the following things need to be done. Regarding Modeling these are:

- Following the guidelines for modeling when creating new models.
- Investigate more efficient ways to facilitate model checking. This could be done by choosing a different modeling strategy with adapted generator according to it, but it could also be done by using different optimization techniques on the obtained mCRL2 specification.

Regarding the Generator these are:

- A “one-click” development integration of model checking in MDSD should be provided.
- The toolchain should be packaged as a tool, so it is usable by Sioux for all projects that work with state machines.

Regarding Property specification these are:

- It should be investigated in which way the properties can be specified in the language of the domain expert or another specification language. For SoLayTec this would be the target language of the Programmable Logic Controller, but for other projects where PLC is not the target, it could be another formalism.
- These properties must automatically be translated in to the μ -calculus or mCRL2, depending on the type of the requirement(s) at hand.

Regarding Model Checking these are:

- For some model checking tasks, scripts or code can be generated, which then can be run (as part of the toolchain) to perform these tasks.
- It should be investigated in which way the formal verification and analysis results will be returned to the domain expert. Message Sequence Charts seem a good option, but also directly relating back to the State Machines will help to use the model checking results to improve the model and thus the system.
- It is useful to further investigate which ways of model checking give the best, i.e., fastest results.

To add real value to Model Driven Software Development, the infrastructure must be applied to more projects and adapted according to the findings as a result of the application.

References

- [1] LWC 2012, *Lwc 2012 - language workbench challenge*, http://www.languageworkbenches.net/index.php?title=LWC_2012, 2012.
- [2] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen, *A tutorial on uppaal*, SFM (Marco Bernardo and Flavio Corradini, eds.), Lecture Notes in Computer Science, vol. 3185, Springer, 2004, pp. 200–236.
- [3] Jan A. Bergstra and Jan Willem Klop, *Process algebra for synchronous communication*, Information and Control **60** (1984), no. 1-3, 109–137.
- [4] Purandar Bhaduri and S. Ramesh, *Model checking of statechart models: Survey and research directions*, CoRR **cs.SE/0407038** (2004).
- [5] Stefan Blom, Jaco van de Pol, and Michael Weber, *Ltsmín: Distributed and symbolic reachability*, CAV (Tayssir Touili, Byron Cook, and Paul Jackson, eds.), Lecture Notes in Computer Science, vol. 6174, Springer, 2010, pp. 354–359.
- [6] components4oaw development, *Ea uml2 exporter build parent -*, <http://components4oaw.sourceforge.net/>, 2012.
- [7] Technische Universiteit Eindhoven, *Home - mcr12 201202.0 documentation*, <http://www.mcr12.org/>, 2012.
- [8] The Eclipse Foundation, *Eclipse - the eclipse foundation open source community website.*, <http://www.eclipse.org/>, 2012.
- [9] ———, *Eclipse modeling project*, <http://www.eclipse.org/modeling>, 2012.
- [10] Blaise Genest and Anca Muscholl, *Message sequence charts: A survey*, ACSD, IEEE Computer Society, 2005, pp. 2–4.
- [11] Jan Friso Groote, Tim W. D. M. Kouters, and Ammar Osaiweran, *Specification guidelines to avoid the state space explosion problem*, FSEN (Farhad Arbab and Marjan Sirjani, eds.), Lecture Notes in Computer Science, vol. 7141, Springer, 2011, pp. 112–127.
- [12] Jan Friso Groote and Mohammad Reza Mousavi, *Modeling and analysis of communicating systems*, <http://www.win.tue.nl/~jfg/educ/2IW26/herfst2011/mcr12-book.pdf>, Okt. 2011.
- [13] Object Management Group, *Omg unified modeling language superstructure version 2.4.1*, <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/>, Aug. 2011.
- [14] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, Jaco van de Pol, and Osmar Marchi dos Santos, *Automated verification of executable uml models*, FMCO (Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, eds.), Lecture Notes in Computer Science, vol. 6957, Springer, 2010, pp. 225–250.
- [15] David Harel, *Statecharts: A visual formalism for complex systems*, Sci. Comput. Program. **8** (1987), no. 3, 231–274.
- [16] David Harel, Robby Lampert, Assaf Marron, and Gera Weiss, *Model-checking behavioral programs*, EMSOFT (Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister, eds.), ACM, 2011, pp. 279–288.
- [17] Matthew Hennessy and Robin Milner, *Algebraic laws for nondeterminism and concurrency*, J. ACM **32** (1985), no. 1, 137–161.
- [18] ISA, *Piping and instrumentation diagrams — isa*, http://www.isa.org/Template.cfm?Section=Distance_Learning1&template=/Ecommerce/ProductDisplay.cfm&ProductID=5785, 2012.
- [19] Peter Kemper and Carsten Tepper, *Traviando - debugging simulation traces with message sequence charts*, QEST, IEEE Computer Society, 2006, pp. 135–136.
- [20] Thomas Klotz, Eva Fordran, Bernd Straube, and Jürgen Haufe, *Formal verification of uml-modeled machine controls*, ETFA, IEEE, 2009, pp. 1–7.
- [21] Johan Lilius and Ivan Paltor, *vuml: A tool for verifying uml models*, ASE, 1999, pp. 255–258.
- [22] Formal Methods and University of Twente Tools, *Ltsmín*, <http://fmt.cs.utwente.nl/tools/ltsmin/>, 2012.

- [23] Inc. Object Management Group, *Object management group*, <http://www.omg.org/>, 2011.
- [24] openArchitectureWare.org, *openarchitectureware.org*, <http://www.openarchitectureware.org/>, 2012.
- [25] Ivan Paltor and Johan Lilius, *Formalising uml state machines for model checking*, UML (Robert B. France and Bernhard Rumpe, eds.), Lecture Notes in Computer Science, vol. 1723, Springer, 1999, pp. 430–445.
- [26] Amalinda Post and Jochen Hoenicke, *Formalization and analysis of real-time requirements: A feasibility study at bosch*, VSTTE (Rajeev Joshi, Peter Müller, and Andreas Podelski, eds.), Lecture Notes in Computer Science, vol. 7152, Springer, 2012, pp. 225–240.
- [27] SoLayTec, *Solaytec - ultrafast spatial ald*, http://www.solaytec.com/downloads/doc_download/3-solaytec-product-brochure, 2012.
- [28] ———, *Ultrafast atomic layer deposition - solaytec*, <http://www.solaytec.com/>, 2012.
- [29] Sparx Systems, *Uml tools for software development and modelling - enterprise architect uml modeling tool*, <http://www.sparxsystems.com/>, 2012.
- [30] R.C. van Cann, *Analysis of stateflow models using mcrl2*, 2008.

A State Machines Meta Models

In this section the meta model of the UML 2.4.1 standard for the State Machine Language is given (Section A.1). The restricted version is given in Section A.2. The notation used in this section is given in Section 2.1.

A.1 UML 2.4.1 State Machines

This section contains a description of the UML State Machine meta model specified by OMG [23] that is relevant to this document. In Figure 20 the meta model is graphically shown. In the following sections, the different classes of the meta model are listed. For the full description of the standard see [13].

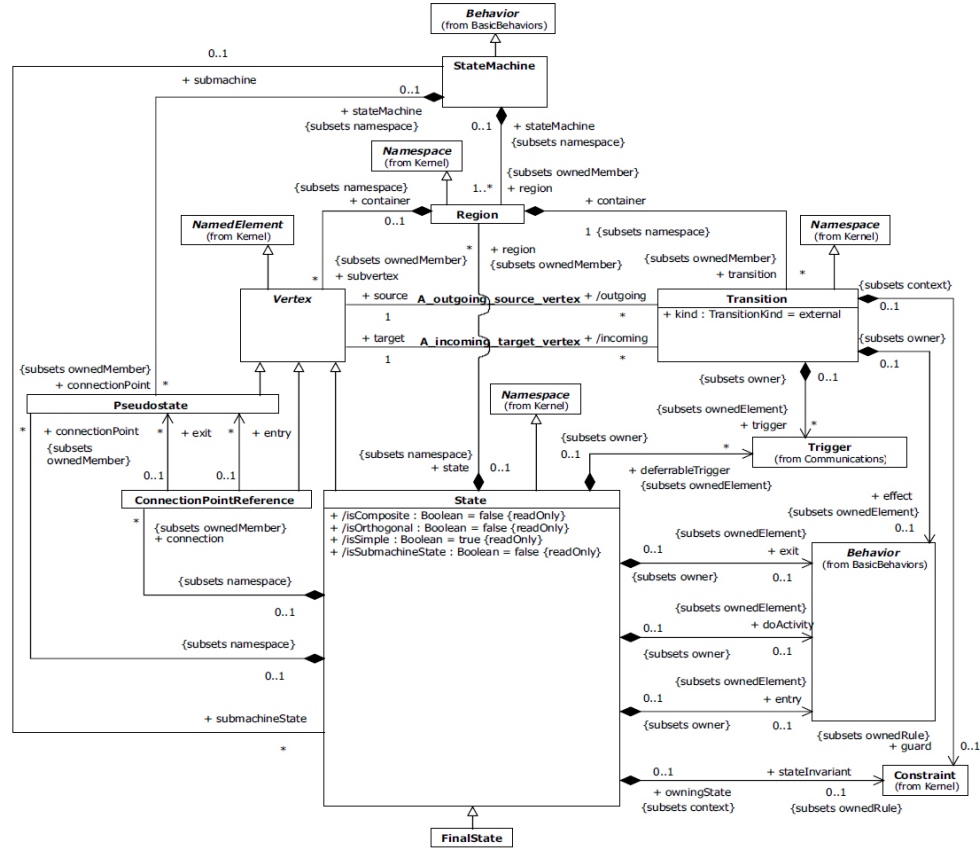


Figure 20: UML 2.4.1 State Machine Meta Model [13, fig. 15.2]

A.1.1 Behavior

Behavior(specification: BehavioralFeature?, context: BehavioredClassifier?, ownedParameter: Parameter, redefinedBehavior: Behavior, precondition: Constraint, postcondition: Constraint, isReentrant: Boolean = true) [13, sec. 13.3.2]

A.1.2 NamedElement

NamedElement(namespace: Namespace?, name: String?, qualifiedName: String?, visibility: VisibilityKind?, inherits: {Element}) [13, sec. 7.3.34]

A.1.3 Namespace

Namespace(elementImport: ElementImport*, importedMember: PackageableElement*, member: NamedElement*, ownedMember: NamedElement*, ownedRule: Constraint*, packageImport: PackageImport*, inherits: {NamedElement}) [13, sec. 7.3.35]

A.1.4 StateMachine

StateMachine(regions: Region⁺, connectionPoint: Pseudostate*, extendedStateMachine: StateMachine*, inherits: {Behavior}) [13, sec. 15.3.12]

A.1.5 Region

Region(stateMachine: StateMachine?, state: State?, transition: Transition*, subvertex: Vertex*, extendedRegion: Region?, redefinitionContext: Classifier, inherits: {Namespace, RedefinableElement}) [13, sec. 15.3.10]

A.1.6 Vertex

Vertex(outgoing: Transition*, incoming: Transition*, container: Region?, inherits: {NamedElement}) [13, sec. 15.3.16]

A.1.7 Pseudostate

Pseudostate(stateMachine: StateMachine?, state: State?, kind: PseudostateKind = initial, inherits: {Vertex}) [13, sec. 15.3.8]

A.1.8 PseudostateKind

Enumeration(initial, deepHistory, shallowHistory, join, fork, junction, choice, entryPoint, exitPoint, terminate) [13, sec. 15.3.9]

A.1.9 ConnectionPointReference

ConnectionPointReference(entry: Pseudostate*, exit: Pseudostate*, state: State?, inherits: {Vertex}) [13, sec. 15.3.1]

A.1.10 State

State(connections: ConnectionPointReference*, connectionPoint: Pseudostate*, deferrableTrigger: Trigger*, doActivity: Behavior?, entry: Behavior?, exit: Behavior?, redefinedState: State?, region: Region*, submachine: StateMachine?, stateInvariant: Constraint?, redefinitionContext: Classifier, isComposite: Boolean = false, isOrthogonal: Boolean = false, isSimple: Boolean = true, isSubmachineState: Boolean = false, inherits: {Namespace, RedefinableElement, Vertex}) [?, sec. 15.3.11]

A.1.11 FinalState

FinalState(inherits: {State}) [13, sec. 15.3.2]

A.1.12 Transition

Transition(trigger: Trigger*, guard: Constraint?, effect: Behavior?, source: Vertex, target: Vertex, redefinedTransition: Transition?, redefinitionContext: Classifier, container: Region, kind: TransitionKind = external, inherits: {Namespace, RedefinableElement}) [13, sec. 15.3.14]

A.1.13 TransitionKind

Enumeration(external, internal, local) [13, sec. 15.3.15]

A.1.14 Trigger

Trigger(event: Event, inherits: {NamedElement}) [13, sec. 13.3.31]

A.1.15 Constraint

Constraint(constrainedElements: Element*, context: Namespace?, specification: ValueSpecification, inherits: {PackageableElement}) [13, sec. 7.3.10]

A.2 Restricted UML State Machines

In this section the restricted State Machine meta model as used in this thesis is described. The listed classes correspond to the meta model given in Figure 5 on Page 14.

A.2.1 Model

Model(inherits: {*Package*})

A.2.2 Package

Package(nestedPackage: *Package**, nestingPackage: *Package*?, class: *Class**, inherits: {*NamedElement*})

A.2.3 Class

Class(isActive: *Boolean* = false, stateMachine: *StateMachine**, package: *Package*, inherits: {*NamedElement*})

When a class has *isActive* = true, then this class is used in the translation.

A.2.4 Behavior

Behavior(inherits: {*NamedElement*})

In Section 4.3.3 the grammar of the effect, entry, do and exit name is described.

A.2.5 Element

Element(owner: *Element*?, ownedElement: *Element**)

A.2.6 NamedElement

NamedElement(name: *String*?, qualifiedName: *String*?, inherits: {*Element*})

A.2.7 StateMachine

StateMachine(region: *Region*, class: *Class*, inherits: {*Behavior*})

A.2.8 Region

Region(stateMachine: *StateMachine*?, state: *State*?, subvertex: *Vertex*⁺, inherits: {*NamedElement*})

A.2.9 Vertex

Vertex(outgoing: *Transition**, incoming: *Transition**, container: *Region*, inherits: {*NamedElement*})

A.2.10 Pseudostate

Pseudostate(kind: *PseudostateKind* = initial, inherits: {*Vertex*})

A.2.11 PseudostateKind

Enumeration(*initial*)

A.2.12 State

State(doActivity: *Behavior*?, entry: *Behavior*?, exit: *Behavior*?, region: *Region*?, isComposite: *Boolean* = false, isSimple: *Boolean* = true, inherits: {*Vertex*})

A.2.13 FinalState

FinalState(inherits: {Vertex})

A.2.14 Transition

Transition(trigger: Trigger?, guard: Constraint?, effect: Behavior?, source: Vertex, target: Vertex, inherits: {NamedElement})

A.2.15 Trigger

Trigger(inherits: {NamedElement})

In Section 4.3.1 the grammar of the trigger name is described.

A.2.16 Constraint

Constraint(inherits: {NamedElement})

In Section 4.3.2 the grammar of the constraint name is described.

B Translation of Restricted UML State Machines to mCRL2

In this section the translation from the restricted UML State Machine meta model (the syntax) from Section A.2 to an mCRL2 Specification is given. The translation process is described in Section 4.

The syntax representation of Section A.2 has an inductive structure. This structure is used to traverse the model which is given as input for the translation. For each element in the meta model a translation function and auxiliary functions are defined. Each section describes a translation function. In each section the return type of the function is denoted. See [12] for their definition.

B.1 Notation

A translation function has the following structure:

$$\llbracket parameters \rrbracket_{ActionNames}^{state} := body$$

where *ActionNames* is a name of the function, *state* denotes the state in which the function is. This is optional. The *parameters* variable represent the parameters which are used in the functions body, denoted by *body*.

When in this body *this* is used, it represents the current element the function is operated on, e.g.,

$$\llbracket Example(parameters), arguments \rrbracket_{Test} := \llbracket this, args \rrbracket_{Example}$$

Here *this* represents *Example(parameters)* and *args* is a subset of *arguments* \cup *parameters* plus possible other elements.

In this example it is also clear that the translation function notation is used in the definition of the function (left) and the calling of other translation functions (right).

The function body has several notation styles, which are explained in the corresponding sections:

- meta notation (Section B.1.1)
- mCRL2 code (Section 2.4), but denoted with **this typeface**.

The elements do not only have the arguments as given in the description of Section A.2, but also attributes (Section B.1.2), auxiliary functions (Section B.1.3) and lists (Section B.1.4).

B.1.1 Meta Notation

$(Child)Parent(parameters)$ denotes casting from the *Parent* class to the *Child* class.

To walk over the inheritance relation the $::$ notation is introduced, so $A::B::C::D::e$ means that *A* inherits from *B* and *B* from *C*, etc. The attribute / association *e* from class *D* is put at the end.

$\langle parameters \rangle$ denotes a tuple.

$\pi_i(\langle parameters \rangle)$ returns the *i*-th parameter of the tuple.

$[]$ denotes the empty list

$[element]$ denotes a singleton list, where *element* is the only element in the list. $element \triangleright list$ denotes a list, where *element* is the head of the list and *list* is the tail of the list.

\emptyset denotes the empty set

$set_1 \cup set_2$ means union of two sets.

$\bigcup_{element \in collection} function(element)$ denotes the union of all the results of $function(element)$ for all the elements of a collection. Note that $function(element)$ should return a set.

$element \in collection$ denotes whether $element$ is included in $collection$. Here $collection$ can be either a set or a list.

$set \leftarrow other_set$ denotes assignment of the right hand side to the left hand side variable. $\{a^i \mid i \in \mathbb{N}\}$ is a set comprehension, which means that this set contains elements a^i for all $i \in \mathbb{N}$. So $i \in \mathbb{N}$ is the domain and a^i is the range.

$expr \text{ whr } a_1 = expr_1, \dots, a_n = expr_n$ is a shorthand notation to avoid duplicate calculations, $expr_i$ may contain variables a_j in the range $1 \leq j \leq i - 1$, while $expr$ may contain variables a_i for $1 \leq i \leq n$.

$a ++ b$ denotes the string concatenation of string a with string b . If a or b is in italics, they denote a variable, and when they are not italics, they denote concrete text.

The construct

```
If condition
  code
Else
  other code
```

denotes a case distinction, where the function definitions, as part of the code, make use of pattern matching.

B.1.2 Attributes

$metaType$ denotes the type of an inherited class. Example: in a parent-child configuration the test ' $parent.metaType = Child$ '

returns $\begin{cases} true & \text{If } Child \text{ is a child of } parent, \text{ i.e., } Child \text{ inherits from } parent \\ false & \text{Otherwise} \end{cases}$

$size$ denotes the number of elements in a collection.

$owners$ is defined on the type $State$ as follows:

```
If Vertex::container.state is available
  Vertex::container.state  $\cup$  Vertex::container.state.owners
Else
   $\emptyset$ 
```

B.1.3 Functions

$selectType(class_name)$ is a function to select elements of a certain type ($class_name$) out of a collection of elements.

$first()$ denotes the first element of a non-empty list, i.e., the head.

$withoutFirst()$ denotes the list without the head, i.e., the tail.

B.1.4 Global sets

$actionNames$ denotes the set of unique action names.

B.2 Translation

In this section is for each syntax element in the State Machine meta model of Section A.2 a translation function defined. The notation used here is explained in Section B.1.

There are different types of generation:

- *single*, where the controller state machines are translated to separate mCRL2 specifications, or
- *full*, where all controller state machines are translated to one mCRL2 specification, where they are put in parallel.

The translation can have two options:

- *simple*, where the guards and the behavior of the states and the transitions is translated to one boolean. This is called serialized.
- *states*, where for each state an action, is introduced, which can be used for reachability check.

If both options are not set, then the variables, which are defined in the UML model, are translated in detail, while the rest of, for example the guard, is omitted. When in a guard or behavior no defined variables occur, it is translated as with option *simple*.

In the sections below it is made clear to which type and option a function belongs.

B.2.1 Model

Returns a set of mCRL2Spec

$$\llbracket \text{Model}(\text{inherits: } \{ \text{Package} \}) \rrbracket_{\text{Model}} := \bigcup_{p \in \text{Package}::\text{nestedPackages}} \llbracket p \rrbracket_{\text{Package}}$$

B.2.2 Package

Returns a set of mCRL2Spec

$$\begin{aligned} & \llbracket \text{Package}(\text{nestedPackage}, \text{nestingPackage}, \text{class}, \text{inherits: } \{ \text{NamedElement} \}) \rrbracket_{\text{Package}} := \\ & \bigcup_{p \in \text{nestedPackage}} \llbracket p \rrbracket_{\text{Package}} \\ & \cup \\ & \bigcup_{c \in \text{class}} \llbracket c \rrbracket_{\text{ActiveClass}} \end{aligned}$$

B.2.3 ActiveClass

Returns a set of mCRL2Spec

$$\begin{aligned} & \llbracket \text{Class}(\text{isActive}, \text{stateMachine}, \text{inherits: } \{ \text{NamedElement} \}) \rrbracket_{\text{ActiveClass}} := \\ & \text{If } \text{isActive} \\ & \quad \{ \llbracket sm \rrbracket_{\text{StateMachine}} \mid sm \in \text{stateMachine} \} \end{aligned}$$

B.2.4 StateMachine

Returns mCRL2Spec

$$\begin{aligned} & \llbracket \text{StateMachine}(\text{region}, \text{class}, \text{inherits: } \{ \text{Behavior} \}) \rrbracket_{\text{StateMachine}} := \\ & \quad \llbracket \text{Behavior}::\text{NamedElement}::\text{name}, \text{region} \rrbracket_{\text{SortSpec}} \\ & \quad ++ \llbracket \text{classNames} \rrbracket_{\text{Elements}} \\ & \quad ++ \llbracket \text{functionUpdates} \rrbracket_{\text{FunctionUpdates}} \\ & \quad ++ \text{sort VarName = struct none} \\ & \quad ++ \llbracket \text{varNames} \rrbracket_{\text{SortSpec VarNames}} \\ & \quad ++ ; \\ & \quad ++ \llbracket \text{region} \rrbracket_{\text{ActSpec}} \\ & \quad ++ \llbracket \text{Behavior}::\text{NamedElement}::\text{name}, \text{this}, \text{region} \rrbracket_{\text{ProcSpec}} \\ & \quad ++ \llbracket \text{Behavior}::\text{NamedElement}::\text{name}, \text{region} \rrbracket_{\text{Init}} \end{aligned}$$

B.2.5 SortSpec

Returns SortSpec

$$\begin{aligned} & \llbracket \text{prefix}, \text{Region}(\text{stateMachine}, \text{state}, \text{subvertex}, \text{inherits: } \{ \text{NamedElement} \}) \rrbracket_{\text{SortSpec}} := \\ & \quad \text{sort E}_+ ++ \text{prefix} ++ _States = \text{struct} ++ \llbracket \text{prefix}, \text{subvertex} \rrbracket_{\text{StateNames}} ++ ; \end{aligned}$$

B.2.6 StateNames

Returns ConstrDeclList

$$\llbracket \text{prefix}, \text{Region}(\text{stateMachine}, \text{state}, \text{subvertex}, \text{inherits: } \{ \text{NamedElement} \}) \rrbracket_{\text{StateNames}} := \llbracket \text{prefix}, \text{subvertex} \rrbracket_{\text{StateNames}}$$

If $\text{vertices} \neq []$

$$\begin{aligned} & \llbracket \text{prefix}, \text{vertex} \triangleright \text{vertices} \rrbracket_{\text{StateNames}} := \\ & \quad \llbracket \text{prefix}, \text{vertex} \rrbracket_{\text{StateName}} ++ \mid ++ \llbracket \text{prefix}, \text{vertices} \rrbracket_{\text{StateNames}} \end{aligned}$$

Else

$$\llbracket \text{prefix}, [\text{vertex}] \rrbracket_{\text{StateNames}} := \llbracket \text{prefix}, \text{vertex} \rrbracket_{\text{StateName}}$$

B.2.7 StateName

Returns ConstrDeclList

```
[[prefix, Vertex(outgoing, incoming, container, inherits: {NamedElement})]]_StateName :=  
  If this.metaType = Pseudostate  
    [[prefix, (Pseudostate)this]]_InitialStateName  
  ElseIf this.metaType = State  
    [[prefix, (State)this]]_StateName  
  ElseIf this.metaType = FinalState  
    [[prefix, (FinalState)this]]_FinalStateName
```

Returns ConstrDeclList

```
[[prefix, State(doActivity, entry, exit, region, isComposite, isSimple, inherits: {Vertex})]]_StateName  
  prefix ++ _ ++ Vertex::NamedElement::name  
  If isComposite  
    ++ | ++[[prefix++ _ ++ Vertex::NamedElement::name, region]]_StateNames
```

B.2.8 InitialStateName

Returns ConstrDecl

```
[[prefix, Pseudostate(kind, inherits: {Vertex})]]_InitialStateName :=  
  prefix ++ _ ++ Vertex::NamedElement::name
```

B.2.9 FinalStateName

Returns ConstrDecl

```
[[prefix, FinalState(inherits: {Vertex})]]_FinalStateName :=  
  prefix ++ _ ++ Vertex::NamedElement::name
```

B.2.10 Elements

```
[[className ▷ classNames]]_Elements :=  
  [[className]]_Element  
  ++ [[classNames]]_Elements
```

```
[[[className]]_Elements] :=  
  [[className]]_Element
```

B.2.11 Element

```
[[className]]_Element :=  
  sort ++ className ++ = struct  
  ++ className ++_default  
  ++ [[className, classInstances]]_Instances  
  ++ ;
```

B.2.12 Instances

```
[[className, classInstance ▷ classInstances]]_Instances :=  
  If className = classInstance.name  
    | ++ classInstance.instance  
  ++ [[className, classInstances]]_Instances
```

```
[[className, [classInstance]]_Instances :=  
  If className = classInstance.name  
    | ++ classInstance.instance
```

B.2.13 FunctionUpdates

```
[[functionUpdate ▷ functionUpdates]]_FunctionUpdates :=  
  [[functionUpdate]]_FunctionUpdate  
  ++ [[functionUpdate]]_FunctionUpdates
```

$\llbracket [functionUpdate] \rrbracket_{FunctionUpdates} :=$
 $\llbracket [functionUpdate] \rrbracket_{FunctionUpdate}$

B.2.14 FunctionUpdate

$\llbracket [functionUpdate] \rrbracket_{FunctionUpdate} :=$
`sort ++functionUpdate ++ = struct`
`++ functionUpdate ++_default`
`++ $\llbracket [functionUpdate, functionValues] \rrbracket_{FunctionValues}$`
`++ ;`

B.2.15 FunctionValues

$\llbracket [functionUpdate, functionValue \triangleright functionValues] \rrbracket_{FunctionValues} :=$
`If functionUpdate = functionValue.name`
`| ++functionValue.value`
`++ $\llbracket [functionUpdate, functionValues] \rrbracket_{FunctionValues}$`

$\llbracket [functionUpdate, [functionValue]] \rrbracket_{FunctionValues} :=$
`If functionUpdate = functionValue.name`
`| ++functionValues.value`

B.2.16 SortSpecVarNames

$\llbracket [varName \triangleright varNames] \rrbracket_{SortSpecVarNames} :=$
`If varName \notin actionNames`
`actionNames \leftarrow {varName}`
`| ++varName ++_var`
`++ $\llbracket [varNames] \rrbracket_{SortSpecVarNames}$`

$\llbracket [varName] \rrbracket_{SortSpecVarNames} :=$
`If varName \notin actionNames`
`actionNames \leftarrow {varName}`
`| ++varName ++_var`

B.2.17 ActSpec

Returns ActSpec*

$\llbracket [Region(stateMachine, state, subvertex, inherits: \{NamedElement\})] \rrbracket_{ActSpec} :=$
`actionNames \leftarrow \emptyset`
 `$\llbracket [subvertex] \rrbracket_{ActionNames}$`
`++ act int;`
`++ act error: Pos`
`++ act Final;`
`actionNames \cup {int, error, Final}`

B.2.18 ActionNames

Returns ActSpec*

$\llbracket [Region(stateMachine, state, subvertex, inherits: \{NamedElement\})] \rrbracket_{ActionNames} :=$
 `$\llbracket [subvertex] \rrbracket_{ActionNames}$`

Returns ActSpec*

`If vertices \neq []`
`$\llbracket [vertex \triangleright vertices] \rrbracket_{ActionNames} :=$`
`$\llbracket [vertex] \rrbracket_{ActionName} ++$`
`$\llbracket [vertices] \rrbracket_{ActionNames}$`
`Else`
`$\llbracket [vertex] \rrbracket_{ActionNames} := \llbracket [vertex] \rrbracket_{ActionName}$`

B.2.19 ActionName

Returns ActSpec*

```
[[Vertex(outgoing, incoming, container, inherits: {NamedElement})]]_ActionName :=  
  If this.metaType = Pseudostate  $\wedge$  outgoing.size > 0  
    [[outgoing]]_ActionName  
  ElseIf this.metaType = State  
    [(State)this]]_StateActionName
```

Returns ActSpec*

```
If transitions  $\neq$  []  
  [[transition  $\triangleright$  transitions]]_ActionName :=  
    [[transition]]_ActionName ++  
    [[transitions]]_ActionName  
  
[[transition]]_ActionName := [[transition]]_ActionName
```

Returns ActSpec*

```
[[Transition(trigger, guard, effect, source, target, inherits: {NamedElement})]]_ActionName :=  
  If trigger is available  
    If [[trigger]]_Trigger  $\notin$  actionNames  
      ++ act ++ [[trigger]]_Trigger ++;  
      actionNames  $\cup$  { [[trigger]]_Trigger }  
  If guard is available  
    If [[guard]]_Constraint  $\notin$  actionNames  
      ++ act ++ [[guard]]_Constraint ++;  
      actionNames  $\cup$  { [[guard]]_Constraint }  
  If effect is available  
    If [[effect]]_Behavior  $\notin$  actionNames  
      ++ act [[effect]]_Behavior ++;  
      actionNames  $\cup$  { [[effect]]_Behavior }
```

B.2.20 StateActionName

Returns ActSpec*

```
[[State(doActivity, entry, exit, region, isComposite, isSimple, inherits: {Vertex})]]_StateActionName :=  
  If entry is available  
    If [[entry]]_Behavior  $\notin$  actionNames  
      ++ act ++ [[entry]]_Behavior ++;  
      actionNames  $\cup$  { [[entry]]_Behavior }  
  If doActivity is available  
    If [[doActivity]]_Behavior  $\notin$  actionNames  
      ++ act ++ [[doActivity]]_Behavior ++;  
      actionNames  $\cup$  { [[doActivity]]_Behavior }  
  If exit is available  
    If [[exit]]_Behavior  $\notin$  actionNames  
      ++ act ++ [[exit]]_Behavior ++;  
      actionNames  $\cup$  { [[exit]]_Behavior }  
  If Vertex::outgoing.size > 0  
    ++ [[Vertex::outgoing]]_ActionName  
  If isComposite  
    ++ [[region]]_ActionNames
```

B.2.21 Trigger

Returns Id

```
[[Trigger(inherits: {NamedElement})]]_Trigger := Parse(trigger ++ NamedElement::name)
```

B.2.22 Constraint

Returns Id

```
[[Constraint(inherits: {NamedElement})]]_Constraint := Parse(constraint ++ NamedElement::name)
```

B.2.23 Behavior

Returns Id

```
[[Behavior(inherits: {NamedElement})]]_Behavior := Parse(effect ++NamedElement::name)
```

B.2.24 ProcSpec

Returns ProcSpec

```
[[prefix, state_machine, Region(stateMachine, state, subvertex, inherits:
  {NamedElement})]]_ProcSpec :=
  ∀class ∈ getClasses()getClassProcess(class)
  actionNames ← ∅
  proc ++ prefix ++( ++
    [[prefix, state_machine, this]]_StateMachineParameters
  ++ ) = ++
  [[prefix, state_machine, [], ∅, this]]_Region ++;
```

B.2.25 StateMachineParameters

Returns IdsDeclList

```
[[prefix, state_machine, Region(stateMachine, state, subvertex, inherits:
  {NamedElement})]]_StateMachineParameters :=
  active_id: Pos ++
  [[prefix, state_machine, this]]_StateMachineParameters2
```

B.2.26 StateMachineParameters2

Returns IdsDeclList

```
[[prefix, state_machine, Region(stateMachine, state, subvertex, inherits:
  {NamedElement})]]_StateMachineParameters2 :=
  ++ , ++prefix++_eCurrentState: E_++state_machine.NamedElement::name++_States
  ++ , ++prefix++_ATransitionWasPerformed: Bool
  ++ [[prefix, state_machine, subvertex]]_StateMachineParameter
```

B.2.27 StateMachineParameter

Returns IdsDeclList

```
If vertices ≠ []
  [[prefix, state_machine, vertex ▷ vertices]]_StateMachineParameter :=
    [[prefix, state_machine, vertex]]_StateMachineParameter
    [[prefix, state_machine, vertices]]_StateMachineParameter
Else
  [[prefix, state_machine, [vertex]]]_StateMachineParameter :=
    [[prefix, state_machine, vertex]]_StateMachineParameter
```

```
[[prefix, state_machine, Vertex(outgoing, incoming, container, inherits:
  {NamedElement})]]_StateMachineParameter :=
  If this.metaType = State
    [[prefix, state_machine, (State)this]]_StateMachineParameter
```

```
[[prefix, state_machine, State(doActivity, entry, exit, region, isComposite, isSimple, inherits:
  {Vertex})]]_StateMachineParameter :=
  If isComposite
    ++ , ++[[prefix++_ ++Vertex::NamedElement::name, state_machine,
    region]]_StateMachineParameters2
```

B.2.28 Region

Returns ProcExpr

```
[[prefix, state_machine, exits, states, Region(stateMachine, state, subvertex, inherits:
  {NamedElement})]]_Region :=
  [[prefix, state_machine, exits, states, subvertex]]_Vertices
```

B.2.29 Vertices

Returns ProcExpr

```
If vertices ≠ []
  [[prefix, state_machine, exits, states, vertex ▷ vertices]]_Vertices :=
    [[prefix, state_machine, exits, states, vertex]]_Vertex
  ++ +
  ++ [[prefix, state_machine, exits, states, vertices]]_Vertices
Else
  [[prefix, state_machine, exits, states, [vertex]]]_Vertices :=
    [[prefix, state_machine, exits, states, vertex]]_Vertex
```

B.2.30 Vertex

Returns ProcExpr

```
[[prefix, state_machine, exits, states, Vertex(outgoing, incoming, container, inherits:
  {NamedElement})]]_Vertex :=
  If this.metaType = Pseudostate
    [[prefix, state_machine, (Pseudostate)this]]_Pseudostate
  ElseIf this.metaType = State
    [[prefix, state_machine, exits, states, (State)this]]_State
  ElseIf this.metaType = FinalState
    [[prefix, state_machine, (FinalState)this]]_FinalState
```

B.2.31 Pseudostate

Returns ProcExpr

```
[[prefix, state_machine, Pseudostate(kind, inherits: {Vertex})]]_Pseudostate :=
  ( ++prefix ++_eCurrentState == ++prefix ++_ ++Vertex::NamedElement::name ++ ) -> (
  If this.Vertex::NamedElement::Element.owner.owner.metaType = StateMachine
    ++ enter_r(active_id) .
  If getOption().contains("states")
    ++ State ++prefix ++_ ++Vertex::NamedElement::name ++ .
  ++ (
    ++ [[prefix, state_machine, [], ∅, Vertex::outgoing]]_Transitions
  ++ )
```

B.2.32 State

Returns ProcExpr

```
[[prefix, state_machine, exits, states, State(doActivity, entry, exit, region, isComposite, isSimple,
  inherits: {Vertex})]]_State :=
  ( ++prefix ++_eCurrentState == ++prefix ++_ ++Vertex::NamedElement::name ++ ) -> (
  If this.Vertex::NamedElement::Element.owner.owner.metaType = StateMachine
    ++ enter_r(active_id) .
  If getOption().contains("states")
    ++ State ++prefix ++_ ++Vertex::NamedElement::name ++ .
  ++ [[prefix, this]]_StateInitPre
  If Vertex::outgoing.size = 0
    ++ [[this]]_StateInitPost
  If Vertex::outgoing.size > 0 ∨ isComposite ∨ doActivity is available
    ++ (
      If exit is available
        [[prefix, state_machine, exit ▷ exits, states ∪ {this},
          this, Vertex::container.state.Vertex::outgoing]]_Transitions
      Else
        [[prefix, state_machine, exits, states ∪ {this},
          this, Vertex::container.state.Vertex::outgoing]]_Transitions
    ++ )
  Else
    ++ delta
  ++ )
```


B.2.33 StateInitPre

```
[[prefix, State(doActivity, entry, exit, region, isComposite, isSimple, inherits:
  { Vertex})]]StateInitPre :=
  If entry is available
    ++ (( ++prefix ++_ATransitionWasPerformed) -> ++[[entry]]Behavior
    If doActivity is available
      ++ . ++[[doActivity]]Behavior ++ <> ++[[doActivity]]Behavior ++ ) .
    Else
      ++ <> int) .
  Else
    If doActivity is available
      ++ [[doActivity]]Behavior ++ .
```

B.2.34 StateInitPost

```
[[State(doActivity, entry, exit, region, isComposite, isSimple, inherits: { Vertex})]]StateInitPost :=
  If ¬isComposite ∧ doActivity is not available ∧ entry is not available ∧ exit is not available
  int
```

B.2.35 StateInit

```
[[Region(stateMachine, state, subvertex, inherits: { NamedElement})]]StateInit :=
  [[subvertex.select(e|e.metaType ≠ FinalState)]]StateInit
```

```
[[vertices]]StateInit :=
  If vertices.size > 0
    [[vertices.first()]]StateInit
    ++ +
    ++ [[vertices.withoutFirst()]]StateInit
  Else
    delta
```

```
[[Vertex(outgoing, incoming, container, inherits: { NamedElement})]]StateInit :=
  ( ++prefix ++_eCurrentState == ++prefix ++_ ++NamedElement::name ++ ) -> (
  If getOption().contains("states")
    ++ State ++prefix ++_ ++NamedElement::name ++ .
  If this.metaType = Pseudostate
    ++ int
  Else
    [[(State)this]]StateInit
  ++ )
```

```
[[State(doActivity, entry, exit, region, isComposite, isSimple, inherits: { Vertex})]]StateInit :=
  [[prefix, this]]StateInitPre
  If isComposite
    ++ [[region]]StateInit
  If isComposite ∧ exit is available
    ++ .
  If exit is available
    ++ [[exit]]Behavior
  ++ [[this]]StateInitPost
```

B.2.36 FinalState

Returns ProcExpr

```
[[prefix, state_machine, FinalState(inherits: { Vertex})]]FinalState :=
  ( ++prefix ++_eCurrentState == ++prefix ++_ ++Vertex::NamedElement::name ++ ) ->
  If this.Vertex::NamedElement::Element.owner.owner.metaType = StateMachine
    ++ enter_r(active_id) .
  If getOption().contains("states")
```

```

    ++ State ++ prefix ++ _ ++ Vertex::NamedElement::name ++ .
  ++ Final . delta

```

B.2.37 Transitions

Returns ProcExpr

```

[[prefix, state_machine, exits, states, pseudoState, transitions]]_Transitions :=
  If transitions ≠ []
  (
    ++ [[prefix, state_machine, exits, transition.first()]]_Transition
    ++ +
    ++ [[prefix, state_machine, exits, states, pseudoState, transitions.withoutFirst()]]_Transitions
    ++ )
  Else
    delta

```

```

[[prefix, state_machine, exits, states, state, transitions]]_Transitions :=
  If transitions ≠ []
  ( ++ [[prefix, state_machine, exits, transitions.first()]]_Transition
    ++ <> tau .
    ++ [[prefix, state_machine, exits, transitions.withoutFirst()]]_Transitions ++ )
  Else
    If state.isComposite
      [[prefix, state_machine, exits, states, state.region]]_Region
    Else
      exit_s(active_id) .
      ++ state_machine.Behavior::NamedElement::name ++ (
        ++ [[state_machine, (target.owners \ source.owners) ∪ target,
          (source.owners \ target.owners) ∪ source]]_NextStateMachineParameters ++ )

```

B.2.38 Transition

Returns ProcExpr

```

[[prefix, state_machine, exits, Transition(trigger, guard, effect, source, target, inherits:
  {NamedElement})]]_Transition :=
  If trigger is available
    ++ sum ++ [[trigger]]_Trigger ++ :Bool . ( ++ [[trigger]]_Trigger ++ ) -> (
  If guard is available
    ++ sum ++ [[guard]]_Constraint ++ :Bool . ( ++ [[guard]]_Constraint ++ ) -> (
  Else
    ++ (true) -> (
  If trigger is available
    ++ [[trigger]]_Trigger ++ .
  If guard is available
    ++ [[guard]]_Constraint ++ .
  If effect is available
    ++ [[effect]]_Behavior ++ .
  If source.metaType = State ∧ ((State)source).isComposite
    ++ ( ++ [[((State)source).region]]_StateInit ++ ) .
    If exits.size > 0
      ++ [[exits, (source.owners \ target.owners) ∪ source]]_Exits
      ++ exit_s(active_id) .
      ++ state_machine.Behavior::NamedElement::name ++ (
      ++ [[state_machine, (target.owners \ source.owners) ∪ target,
        (source.owners \ target.owners) ∪ source]]_NextStateMachineParameters
      ++ )
    Else
      If exits.size > 0
        [[exits, (source.owners \ target.owners) ∪ source]]_Exits
      ++ exit_s(active_id) .
      ++ state_machine.Behavior::NamedElement::name ++ (

```

```

    ++ [[state_machine, (target.owners \ source.owners) \cup target,
      (source.owners \ target.owners) \cup source]]_NextStateMachineParameters
    ++ )
  If trigger is not available \vee guard is not available
    ++ )
  Else      ++ ))

```

B.2.39 Exits

Returns ProcExpr

```

[[exit \triangleright exits, states]]_Exits :=
  If exits \neq []
    If exit.NamedElement::Element::owner \in states
      [[exit]]_Behavior ++ .
    ++ [[exits, states]]_Exits

[[[exit], states]]_Exits :=
  If exit.NamedElement::Element::owner \in states
    [[exit]]_Behavior ++ .

```

B.2.40 NextStateMachineParameters

Returns DataExprList

```

[[state_machine, targets, sources]]_NextStateMachineParameters =
  [[state_machine.Behavior::NamedElement::name, state_machine,
    state_machine.region, targets, sources]]_NextStateMachineParameters2,
  \pi_1([[state_machine.region]]_NextGuardValues) \pi_1([[subvertices, \emptyset, "", next]]_BehaviorVariables)

```

B.2.41 NextStateMachineParameters2

Returns DataExprList

```

[[prefix, state_machine, Region(stateMachine, state, subvertices, inherits : {NamedElement}),
  targets, sources]]_NextStateMachineParameters2 =
  [[state_machine.Behavior::NamedElement::name, state_machine, subvertices,
    targets, sources]]_NextCurrentStateName,
  [[prefix, state_machine, subvertices, targets, sources]]_NextStateMachineParameter

```

B.2.42 NextStateMachineParameter

Returns DataExprList If vertices \neq []

```

[[prefix, state_machine, vertex \triangleright vertices, targets, sources]]_NextStateMachineParameter =
  [[prefix, state_machine, vertex, targets, sources]]_NextStateMachineParameter
  [[prefix, state_machine, vertices, targets, sources]]_NextStateMachineParameter

```

Else

```

[[prefix, state_machine, [vertex], targets, sources]]_NextStateMachineParameter =
  [[prefix, state_machine, vertex, targets, sources]]_NextStateMachineParameter

```

```

[[prefix, state_machine, Vertex(outgoings, incomings, container, inherits : {Named-
  Element}), targets, sources]]_NextStateMachineParameter =

```

If Vertex.type = State

```

  [[prefix, state_machine, (State)this, targets, sources]]_NextStateMachineParameter

```

```

[[prefix, state_machine, State(doActivity, entry, exit, region, isComposite, isSimple,
  inherits : {NamedElement, Vertex}), targets, sources]]_NextStateMachineParameter =

```

If isComposite

```

  , [[prefix, state_machine, region, targets, sources]]_NextStateMachineParameters2

```

B.2.43 NextCurrentStateName

Returns DataExpr

```

[[prefix, state_machine, vertex \triangleright vertices, targets, sources]]_NextCurrentStateName =

```

```

If  $targets \cap sources = \emptyset$ 
   $\llbracket prefix, state\_machine, vertex \triangleright vertices, targets, sources \cup$ 
   $sources.childs \rrbracket_{NextCurrentStateName}$ 
If  $vertices.size \geq 0$ 
  If  $Vertex.type \neq Pseudostate$ 
    If  $vertex$  in  $targets$ 
       $prefix \# state.name$ 
      , true
    ElseIf  $vertex.container.state$  in  $sources$ 
       $\llbracket prefix, vertex.container.subvertices \rrbracket_{InitialStateName}$ 
    Else
       $\llbracket prefix \# vertex.NamedElement.name, state\_machine, vertices, targets,$ 
 $sources \rrbracket_{NextCurrentStateName}$ 
  Else
     $prefix \# CurrentState$ 
    , false

```

B.2.44 Init

Returns Init

```

 $\llbracket prefix, Region(stateMachine, state, subvertices, inherits : \{NamedElement\}) \rrbracket_{Init} =$ 
proc  $Coordinator(active\_id : Pos) = enter\_s(active\_id).exit\_s(active\_id).$ 
 $Coordinator(if(active\_id == 1, 1, active\_id + 1));$ 
init  $comm(\{exit_s | exit_r \rightarrow exit, \dots\}, prefix(\llbracket prefix, this \rrbracket_{InitialStateMachineParameters} || Var(\dots) ||$ 
 $Coordinator(1);$ 

```

B.2.45 InitialStateMachineParameters

Returns DataExprList

```

 $\llbracket prefix, Region(stateMachine, state, subvertices, inherits : \{Namespace, Redefinable-$ 
 $Element\}) \rrbracket_{InitialStateMachineParameters} =$ 
 $\llbracket prefix, this \rrbracket_{InitialStateMachineParameter2}$ 
 $\pi_1(\llbracket prefix, subvertices \rrbracket_{InitialGuardVariables}) \quad \pi_1(\llbracket subvertices, \emptyset, "", init \rrbracket_{BehaviorVariables})$ 

```

B.2.46 InitialStateMachineParameters2

Returns DataExprList

```

 $\llbracket prefix, Region(stateMachine, state, subvertices, inherits : \{Namespace, Redefinable-$ 
 $Element\}) \rrbracket_{InitialStateMachineParameters2} =$ 
 $\llbracket prefix, subvertices \rrbracket_{InitialStateName},$ 
 $\llbracket prefix, subvertices \rrbracket_{InitialStateMachineParameter}$ 

```

B.2.47 InitialStateMachineParameter

Returns DataExprList

```

If  $vertices \neq []$ 
   $\llbracket prefix, vertex \triangleright vertices \rrbracket_{InitialStateMachineParameter} =$ 
   $\llbracket prefix, vertex \rrbracket_{InitialStateMachineParameter}$ 
   $\llbracket prefix, vertices \rrbracket_{InitialStateMachineParameter}$ 
Else
   $\llbracket prefix, [vertex] \rrbracket_{InitialStateMachineParameter} =$ 
   $\llbracket prefix, vertex \rrbracket_{InitialStateMachineParameter}$ 

 $\llbracket prefix, Vertex(outgoings, incomings, container, inherits : \{Named-$ 
 $Element\}) \rrbracket_{InitialStateMachineParameter} =$ 
  If  $Vertex.type = State$ 
     $\llbracket prefix, (State)this \rrbracket_{InitialStateMachineParameter}$ 

 $\llbracket prefix, State(doActivity, entry, exit, region, isComposite, isSimple, inherits : \{Named-$ 
 $Element, Vertex\}) \rrbracket_{InitialStateMachineParameter} =$ 

```

If *isComposite*
, $\llbracket \text{prefix} ++ \text{Vertex}::\text{NamedElement}::\text{name}, \text{region} \rrbracket_{\text{InitialStateMachineParameters2}}$

B.2.48 InitialStateName

Returns DataExpr

If *vertices* $\neq []$

$\llbracket \text{prefix}, \text{vertex} \triangleright \text{vertices} \rrbracket_{\text{InitialStateName}},$
 $\llbracket \text{prefix}, \text{vertex} \rrbracket_{\text{InitialStateName}}$
 $\llbracket \text{prefix}, \text{vertices} \rrbracket_{\text{InitialStateName}}$

Else

$\llbracket \text{prefix}, [\text{vertex}] \rrbracket_{\text{InitialStateName}} =$
 $\llbracket \text{prefix}, \text{vertex} \rrbracket_{\text{InitialStateName}}$

$\llbracket \text{prefix}, \text{Vertex}(\text{outgoings}, \text{incomings}, \text{container}, \text{inherits} : \{\text{NamedElement}\}) \rrbracket_{\text{InitialStateName}} =$

If *Vertex.type* = *Pseudostate*

$\llbracket \text{prefix}, (\text{Pseudostate})\text{this} \rrbracket_{\text{InitialStateName}} =$

$\llbracket \text{prefix}, \text{Pseudostate}(\text{kind} : \text{PseudostateKind} = \text{initial}, \text{inherits} : \{\text{Vertex}\}) \rrbracket_{\text{InitialStateName}} =$

$\text{prefix} ++ \text{Vertex}::\text{NamedElement}::\text{name}$

, **true**

C Toolsets

In this section the toolsets that are used in this thesis and work with mCRL2 in- and output are described. Section C.1 gives an overview of the mCRL2 toolset version 201202.0.10804 (Release) that is found at [7]. In Section C.2 a part of the LTSMIN toolset is described. The toolset is found at [22] and the theory behind is described in [5].

C.1 mCRL2 Toolset Overview

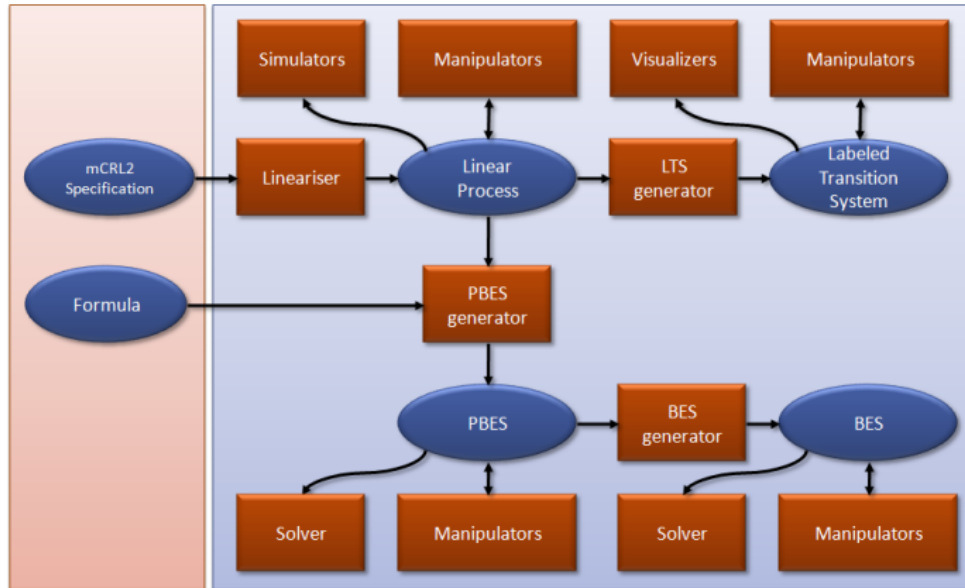


Figure 21: Toolset Overview (from [7])

The mCRL2 Specification is a plain-text file which specifies the behavior of the system that will be analyzed in the mCRL2 language (see Section 2.4). The Linearizer `mcr122lps` is a tool that takes an mCRL2 Specification and transforms it into an LPS. In an LPS all the parallelism is removed.

C.1.1 LPS Tools

As soon as the Specification is transformed into the LPS format several tools are available to start analyzing. The first is simulating the model, i.e., from the initial state performing sequences of actions. Hereby unexpected or erroneous behavior can be observed. The tool `lpsxsim` provides a graphical user interface for simulation.

`lpsinfo` provides statistical information about an LPS.

`lpspp` can print the binary format of an LPS in a humanreadable format.

`lps2lts` can generate the state space (LTS) from an LPS, since the LPS is a symbolic representation of an LTS, which describes the behavior of the system explicitly.

State space generation can take a lot of time and therefor there are tools that can reduce or alter the LPS to make it more suitable for state space generation. The tool

`lpssumelm` eliminates summands that are redundant;

`lpssuminst` instantiates summands, so expanding them for every data value in it;

`lpsconstelm` removes constant variables from the specification, by substituting the values for them;

`lpsparelm` removes parameter of the process specification, that do not contribute to the behavior of the system;

`lpsrewr` rewrites data expressions to their normal form;

`lpsconfcheck` marks confluent tau's.

`lpsrealelm` removes real variables.

C.1.2 LTS Tools

When an LTS is generated, other tools can be used to visualize the state space. A graph representation is provided by `ltsgraph`. For small state spaces this tool is suitable, but for larger ones another visualization tool can be used: `ltsview`. This tool clusters the nodes, so the complexity of the image is reduced. It produces a 3D visualization which aims to show symmetry in the behavior of the system. The tool `diaggraphica` clusters the states also and depicts them in a 2D representation. It clusters based on state parameters, while `ltsview` clusters on structural properties. Besides visualization, reduction modulo several equivalences, belongs to the options. The tool `ltsconvert` often reduces the state space dramatically, preserving important properties. For example, if there are a lot of internal actions, these can be removed, and thus reduces the state space. Some equivalences reduce more, but also at the cost of losing some properties. The tool can also convert between several textual and binary LTS file formats. Another tool, `ltscompare`, can compare two LTSs to check whether they are behaviorally equivalent or similar given several notions of equivalence or similarity. An LTS can also be transformed back to an LPS using `lts2lps`, for example to further analyze the specification with LPS tools after minimization of the LTS.

C.1.3 PBES Tools

To verify that certain desired or undesired properties respectively hold or not hold, the aforementioned tools give insight into the behavior of the modeled system, but cannot give an answer to complex model checking questions. In the mCRL2 toolset model checking is done by parameterized boolean equation systems (PBESs). Model checking starts mostly with an LPS (or LTS) and a formula expressing the property that must hold or must not hold. This formula is expressed using the regular modal μ -calculus (extended with data). (See also Section 2.5). The tool `lps2pbes` takes such LPS and formula and produces a PBES. Solving the PBES yields an answer, but that is generally undecidable, and thus may fail. The main tool for trying to solve a PBES is `pbes2bool`. There are other tools that help getting insight into the PBES and also reduce it. The tool

`pbespp` prints the binary PBES in human readable format;

`pbesinfo` gives some statistical information about the PBES;

`pbesrewr` simplifies a PBES, by rewriting data expressions;

`pbesconstelm` removes constant variables from the PBES, by substituting the values for them;

`pbesparelm` removes parameters that do not contribute to the solution of the PBES;

`pbesabstract` abstracts from given variables, making the PBES less true, but if the answer is still true, then the original PBES certainly was true also.

The toolset also contains some import and export tools, which are not used for this project.

C.2 LTSMIN Toolset

The toolset consists of tools, which can manipulate and model check Labeled Transition Systems. The tool used in this thesis is `lps-reach`. This tool performs symbolic reachability analysis for mCRL2 models. The input of the tool is a Linear Process Specification generated with `mcr122lps` (see Section C.1).