TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

The semantics of ALIAS defined in mCRL2

Jonk, R.J.W.

Award date:
2016

[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

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

The semantics of ALIAS defined in mCRL2

Master Thesis

Ruben J. W. Jonk

Supervisors:
prof.dr.ir. Jan Friso Groote
ir. Dennis Hendriks, ASML
dr.ir. Tom Verhoeff

Eindhoven, 25 August 2016

Abstract

Separating systems into data, algorithmic and control components is an important step in making systems more reliable and easier to maintain. Dividing a system into smaller components reduces the complexity in each separate component. The control language in which these components are designed at ASML is called ALIAS.

A control system in this work is characterized by the use of state machines. The control flow and decisions an ALIAS component can make is decided by the states and transitions designed by the creator of the component. These state machines can be described by labeled transition systems. By defining a translation from ALIAS to such labeled transitions systems we formalized the semantics of the control language.

This translation of state machines is accomplished by translating the state machine of the control component to an mCRL2 specification. mCRL2 is a formal model checker which allows us to create and analyze labeled transition systems. The translation is built in QVTo, a domain specific language developed to express and perform model-to-model transformations. The validity of the transformation is checked by transforming a set of production models from ASML and checking the resulting transition systems.

The results of the transformation show that the ALIAS models can be translated to an mCRL2 specification. These models are transformed into labeled transition systems. The results show that the transformation to mCRL2 is feasible. Furthermore, a set of manually created models to validate the transformation showed positive results: the resulting output transition systems match the expected input models.

The prototyping done in this work shows that mCRL2 is a feasible approach to transform ALIAS into another formalism and defining the semantics of ALIAS. The prototyping work is however not a finished product and improvements can be made on the implementation of the tool to transform and validate the correctness of the translation.

Contents

Chapter 1

Introduction

Complex systems can be decomposed in various types of components [?]. The first type of components are data components, which store and provides access to sets of data. The second type of components describes algorithms, components that require a significant amount of time in the form of computing power or controlling actuators to perform actions. The final, third type of components comprise of the control components: top level components that orchestrate the algorithmic and data components and provide a control flow: what data and algorithm components are used and in which order are they being used.

The scope of this work is limited to control components. In particular, it is a prototyping work to explore the possibility of translating a new control specification language to a formal model language to define the semantics of the control language. An example of this semantics is defining the way how the control language shows and/or deals with unwanted behaviour of the component, such as an “illegal” action. Control component models that are transformed to the formal model language are transformed in a labeled transition system based on the specification in the formal model. These labeled transitions systems are then validated by inspecting the transition system and compare it with the expected behaviour as defined by the semantics.

Model transformations are a key concept in model-driven engineering and automating the process of transforming one model into another model and vice versa. For instance, model transformation was done in [?] to show the value of model transformation in the automotive industry. In this work, the purpose of the model transformation is to introduce semantics to the source model by translating the models to formal model language.

To illustrate the various components in a system, we provide a small example. A simple control flow can describe the following case: a new wafer is loaded on the track, and the wafer data is collected. Following this step, the wafer is calibrated and eventually the data is updated. The wafer is then passed on to the next component. Figure ?? shows this simple control flow. The control flow consists of four states (rectangles) and every state performs an action on the components it controls. After taking in the wafer, the wafer data is requested. This information is then used to perform a calibration on the wafer before the control system outputs the wafer. The example shows how data is modeled as a separate component, and used to perform an algorithm while calibrating the wafer. Furthermore, the wafer is used as input and output and needs to be transported physically, and these actions can be considered as durative actions and are part of the algorithm component.

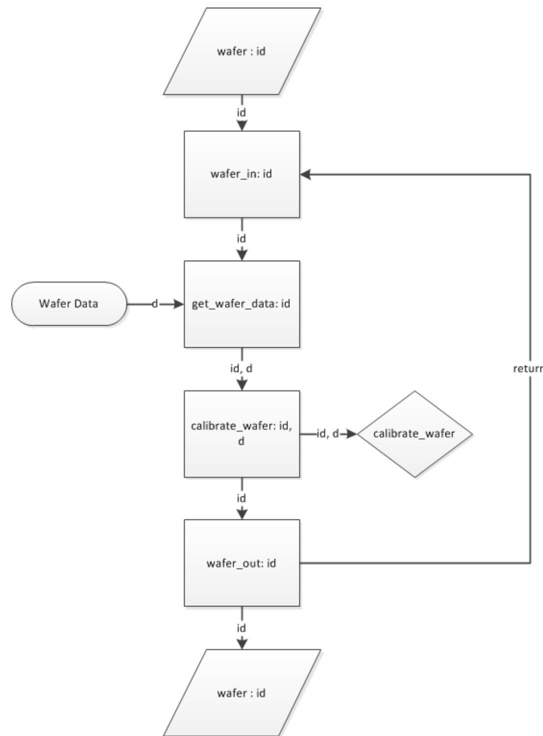

Figure 1.1: A control flow of a simple process.

1.1 ALIAS

ALIAS is the ASML language to realize the control component of the example described above. Models written in ALIAS are decomposed in small components which can be individually modeled, reducing the complexity of each component. A short overview of the most important concepts in ALIAS are summarized below.

- Hierarchy of components — Every component in ALIAS consists of a single building block and interacts with the system it is part of through its provided interface(s), which declares how the component should be used, and its required interface(s), the components it uses. The composition of these components forms a hierarchy in a tree structure.

- Operations, Replies, Notifications — Components interact with other components using messages of various types. Operations and Reply messages are used when a component is invoking an action of a lower level component. A lower level component can notify a higher level component that an event happened by means of a notification message. Operations and Replies are synchronous calls, while a notification is decoupled and thus asynchronous, by means of a queue.

- State Machine — The behaviour of a component is described by a state machine. As an illustration, a Mealy machine [?] is a simple version of the type of state machines that are being used for ALIAS. The output of a Mealy machine depends on the current state and the current input. This is in contrast with Moore machines, where the output only depends on the current state.

1.2 The formal model language: mCRL2

mCRL2 [?] is a formal model verification tool. Specifications written in mCRL2 are based on process algebra and can be transformed into labeled transition systems. The toolset is used to give a meaning to the processes of ALIAS. Process algebra is a tool suitable to define execution order and flow control, which are two of the most important concepts in ALIAS. Therefore, mCRL2 is used to give concrete semantics to these concepts.

1.3 Approach

An overview of the artifacts that are involved in the prototyping work is shown in Figure ??. Dashed lines indicate these objects are provided by ASML or are already otherwise available. At the left are the ALIAS objects that are supplied by ASML. The mCRL2 toolset is also a supplied toolset ready to be used.

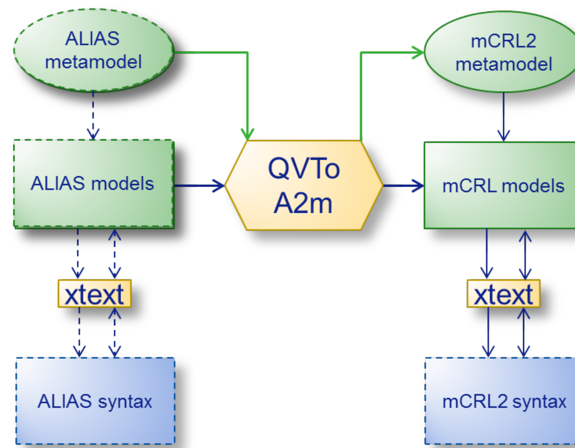

Figure 1.2: A schematic view of the relevant parts of the work. Dashed lines indicate these parts are provided by ASML or existing tools, solid lines indicate deliverables.

There is no suitable meta-model [?] available for the mCRL2 toolset. The importance of this meta-model is clear by the translation method: a QVTo [?] model-to-model mapping from ALIAS to mCRL2. An Xtext [?] model grammar is used as the link between the concrete mCRL2 syntax and mCRL2 models. By using an Xtext grammar, based on the provided grammar for the mCRL2 language, an mCRL2 meta-model is automatically generated. Whilst there are drawbacks to using this approach, such as a meta-model with redundant classes, for the scope of this project a generated mCRL2 meta-model containing only the relevant parts of mCRL2 for this prototype is sufficient.

1.3.1 Outline

The outline of this report is as follows. First, the concepts and background of ALIAS and mCRL2 are described in Chapter 2. We elaborate on the concepts outlined in the description of ALIAS and introduce more in-depth concepts. Following that, in Chapter 3 we discuss the methodology of the prototyping work: which steps have been taken to complete the translation of ALIAS to mCRL2 and how the models themselves are verified. Chapter 4 provides a description on the translation to mCRL2 and the parts of ALIAS being translated. In Chapter 5 we follow with a validation of the translation by using handcrafted models as well as a set of models provided by ASML. Finally, we conclude with our findings in Chapter 6.

Chapter 2

Language Concepts

This section explains the key concept of the work. The background of the work plays a key role in the understanding and translation of ALIAS, the new modelling language for ASML as a part of ASOME: ASML SOftware Modeling Environment.

2.1 Background: Control Verification Frameworks

The purpose of ALIAS can be described by the following three use cases:

1. ALIAS is used as a control specification language. It is used to create technology independent control models and used as a bridge to state of the art control verification framework(s). Such a framework has a code generator that guarantees correct implementation of the models written in that verification framework. For this bridge between ALIAS and the verification framework, these semantics have to be formalized. ASML uses ASD [?] as the control verification framework.[1]

2. ALIAS is used as a connection of control models to the data and algorithm models.

3. ALIAS is used to automatically generate the glue code to embed the generated control code into existing ASML code.

Models in the control verification framework(s) are called control models. Control models are a type of state machine and allow the designer to control the order in which actions can occur, using Sequence-Based Specification (SBS) rules. These types of models form the basis for this work.

The user develops a model using the control model development environment. The toolset available in such a control verification framework allows the user to formally verify a number of properties, e.g. absence of deadlock or illegal actions, by means of generating a formal model and running the verification procedure. This verification procedure is at the back-end of any control verification framework.

Once the verification procedure returns no errors and the control model is deemed defect free, defect free code can be generated from the developed model. The control verification framework only then guarantees that the formal model without design errors is equivalent to the generated source code.

A control specification consists of multiple components. At every level of the decomposition are the interface and design model components. Each of these components contain a state machine, consisting of states and transitions with events (triggers) and actions (compare with input in Mealy machines) performed on a transition which describe the behaviour of the component. These components are explained in more detail in the following sections.

[1] Analytical Software Design (ASD) is a product from Verum and is such a control verification framework. ASD is used to automatically generate source code in Java, C, C++ or C# based on models built in the ASD environment.

2.1.1 Specification and implementation models

Control models come in two types of models: specification models and implementation models. Specification models describe how a component should behave. A specification model contains a set of events or operations a client using a component can trigger. Associated with each possible event, the interface model specifies whether this event is legal or illegal. In the case of legal events, there may be a sequence of operations or no operation to be done at all, depending on the internal state of the interface. In the case of a legal event, an appropriate reply is sent back to the client that uses the interface. Furthermore, a specification specifies which notifications a client of a component can expect to receive.

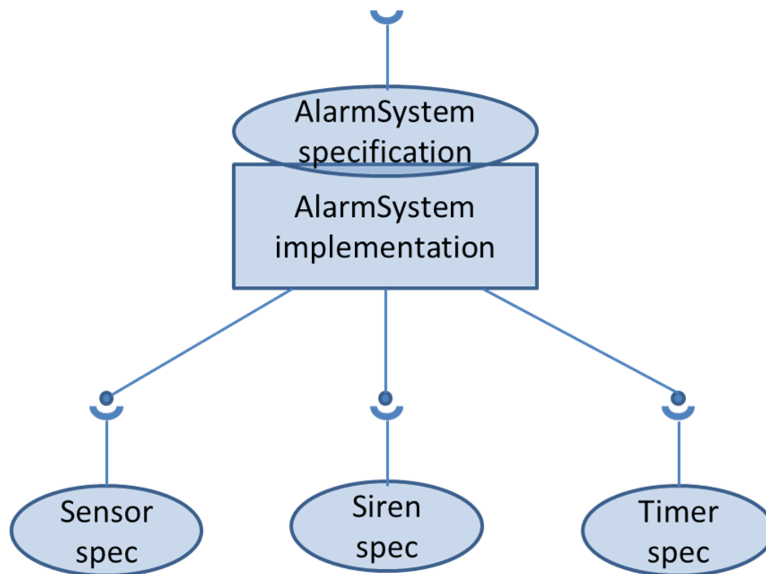

Figure 2.1: A view of the hierarchy of specification and implementation models.

The implementation model is an implementation of a specification model. Where the specification model defines how a component is expected to behave from an outside point of view, e.g. a user using the interface, the implementation model is the implementation of a component. An implementation model may require the use of lower level components through their specifications; it then acts as a user of those components and is required to adhere to the specifications of the models it uses. A component that is being used by a higher level component is called a server or service.

As an example, the alarm system model is used to illustrate the process. Figure **??** shows the schematic view of the alarm system and its components. The implementation model of the alarm system has an associated specification model for a client of the alarm system to implement. This client is unaware of the layers below the alarm system; only the provided interface of the alarm system is relevant.

The specification model of the alarm system uses three other interfaces: a sensor, a timer, and a siren. The sensor is used to detect movement. Upon a detection of movement, a timer is started using the timer interface. The timer is used to give the user time to disable the alarm before the siren eventually goes off. This implementation of the alarm system is not unique: a variant of the alarm system could include a version without the timer component, i.e. the siren is enabled immediately. This means that a specification may be reused to create different components adhering to the same specification.

2.1.2 States and transitions

Control models describe state machines. A model consists, amongst others, of states and transitions. A transition typically consists of at least an event and an action. An action can be a sequence of actions, the illegal action or the no operation action (empty action). A target state is specified to indicate the next state of the model if the transition is not declared illegal. Transitions contain guards and updates on state variables to allow a designer to control the flow of the component. Each transition is specified by an SBS rule. This rule describes the event, the actions and how to proceed to the next state. A state may contain an invariant, an expression using state variables that is required to evaluate to true at all time while being in that state.

An event can be one of three types: 1) a method call to a client using the interface from a higher level, 2) a notification from a used interface at a lower level (this is called a server or service) and 3) a modeling event (interface model only). The modeling event for interface models describes internal behaviour of the component implementing the interface.

The second element of a transition is the actions. Actions consist of invoking operations on components at a lower level, sending notifications to a component at a higher level and sending replies to users of the components. A method call is invoked on a component at a lower level and tells the component to perform a number of actions. For each method call, the component is required to generate a reply action to signal the caller of the method it has finished executing the method call. These reply actions may either by 'void' or contain a value; reply actions are thus either void replies or valued replies. A notification action sent to a client is handled by placing the notification in the queue of the client. The purpose of notifications is to signal the client it has completed it's tasks.

Method calls are called synchronous messages between components. A method call or operation invocation on a server is run until completion; that is, a caller will block until the callee sends a reply back to the caller such that it can continue its own process. Notifications are sent to a queue, causing a decoupling between lower level components and higher level components. This kind of messages are called asynchronous messages between components. The queue has priority over a new method call from the client using the component. The queue is blocked from being read while the client is using the component, however. This is called run to completion. The queue is polled for the presence of messages, and only when the queue is empty, may the client invoke a new method call.

A sequence of operations consists of method calls, outgoing notifications and a (mandatory) reply when the event is a method call. A method call is the counterpart of the method-call event (i.e. a method call acts as an event to a method-call event), while a notification read from the queue is the counterpart of a raised notification. A caller is blocked until the callee notifies the caller it is done executing its rule case using the reply action. This reply action may occur later in the multi-threaded execution model, as part of the SBS rules from an incoming notification from a used service.

An abstract representation of a transition from one state to another is shown in Figure **??**. A set of variables is declared globally. These variables are used to evaluate the invariant of the state and guards on each transition. Before entering the next state, these variables are updated. A transition itself consists of a trigger and a guard to enter the transition. The actions associated with the transition are then executed before the next state is entered.

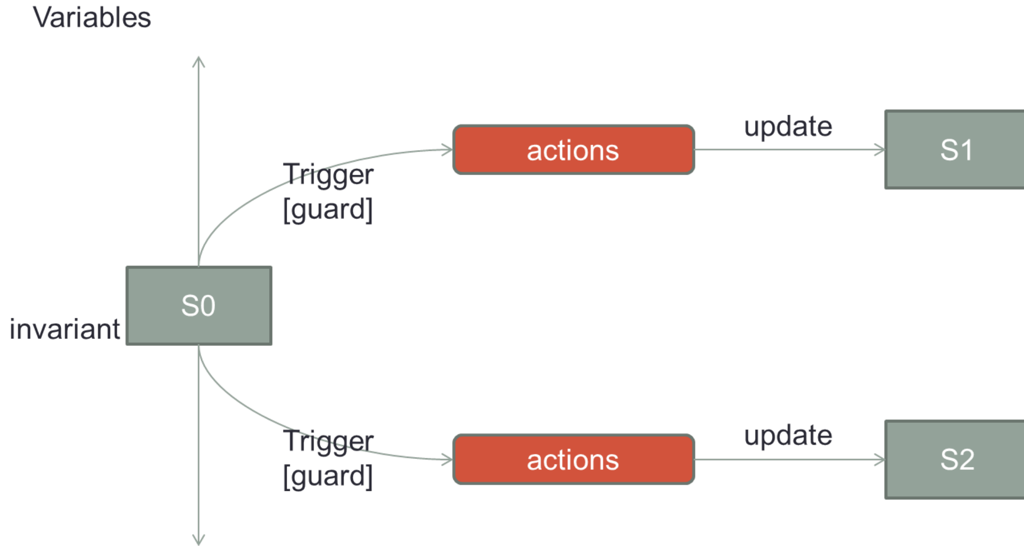

Figure 2.2: A transition with its abstract elements.

2.2 Single-threaded and multi-threaded execution

In ALIAS , models may be deployed using single- or multithreaded semantics. In the single-threaded semantics, there is only one thread of control, wheras in multithreaded execution semantics, every realization model is assigned a thread. The detailed description of these different execution semantics is not further discussed in this work and we refer to the full version of the report for a more detailed description.

F

2.3 mCRL2

mCRL2 is the target language of the translation. mCRL2 is a language that allows you to write process algebra. Process algebra is a convenient way to describe sequences of actions in a structured way. These specifications are converted to state spaces.

State machines in ALIAS are translated to process expressions in mCRL2. The syntax for process expressions is given below; however limited to the used operators in the translation. The syntax of an mCRL2 process expression is given by the following grammar:

P =	
a	Multi-action
| Q	Process
| P + P	The choice operator
| P . P	The sequential operator
| P | P	The communicate operator
| P || P	The parallel operator
| (g) ->P	If statement
| sum d:D . P	The sum operator
| (P)	Brackets

where a is a multi-action, Q is a process. g is a boolean expression, d an mCRL2 data variable and D an mCRL2 data type. The choice operator allows for a process expression to make a choice between the left side and the right side of the operator. A process with a sequence of actions is denoted by the sequential operator: the actions are executed from left to right.

The communicate operator is used to force the process to proceed by performing both actions, at the same time. The parallel operator instead allows the left process to proceed the right process or vice versa. Executing both the left and right sides of the parallel operator at once is also allowed.

The if-statement provides a conditional on the process. Only if the guard g is true may the process expression be executed. In the translation, only the 'if' part is used, the 'else' part is not used and thus omitted from the syntax. The sum operator is used to generate a variable d of data type D, which may be used in actions and processes.

For a full description of the mCRL2 toolset, see [?].

Chapter 3

Methodology

In order to provide a translation from ALIAS to mCRL2, we first define the elements that are already present, what we work towards and the steps that need to be done to achieve the goal. A schematic overview of the project is shown in Figure ??. In this figure, dashed lines indicate elements that are already present or provided by ASML. Solid lines indicate components that are created as part of the project to translate ALIAS to mCRL2.

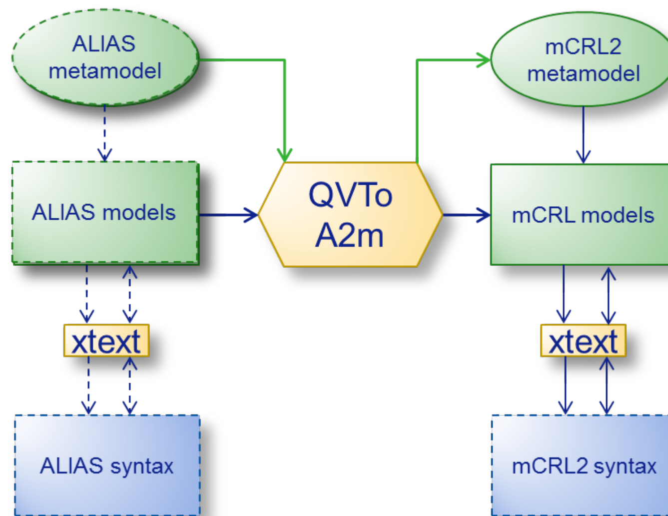

Figure 3.1: A schematic view of the relevant parts of the work. Dashed lines indicate these parts are provided by ASML or existing tools, solid lines indicate deliverables.

In this chapter, the building blocks of the project are described. For each block that is created in this project, a description on how it is created is included. First, an overview of the used environment is given.

3.1 Environment

The ASOME project is developed in Eclipse version Mars.2, an integrated development environment (IDE). Eclipse supports various tools and an interface to develop software and models. The most important tools supported by the Eclipse IDE which are used for this project are the following:

Eclipse Modelling Framework

The Eclipse Modelling Framework (EMF) is a framework that allows the creation of structured data-models, i.e. meta-models. Knowledge of meta-models is assumed for this work, [?] provides a good overview of the meta-model concept. Meta-models, defined in Ecore files, can be used to generate various artifacts such as a creation wizard for models.

Xtext

Going from a model to text requires a model-to-text transformation. Going from text to a model requires a parser to read the input tokens and construct the model. Xtext is a framework that allows you to create a parser which can read text and construct the model, and vice versa. To read more on Xtext, we refer to [?].

Xtext is used to give a concrete syntax to the ALIAS meta-model that is being developed. However, Xtext can also be used to define a grammar for a domain specific language and then generate a meta-model based on the specified grammar. This meta-model can then be used to generate other artifacts, such as an EMF model which can then be used to create a text editor for the specified grammar. This is the approach taken to create the mCRL2 meta-model and how the concrete syntax of ALIAS is defined.

QVTo

Query View Transformation operational (QVTo) is a framework to define model-to-model transformations at a meta-model level. These transformations are then applied to concrete models belonging to such a meta-model and transformed into a model of the target meta-model. To read more on QVTo model transformations, we refer to [?].

3.2 ALIAS

The leftmost side in Figure ?? indicates the ALIAS part. For the scope of this project, the ALIAS meta-model is of importance for creating the model-to-model transformation to mCRL2. The ALIAS models, which conform to the ALIAS meta-models, are used as input for the transformation.

For the project, the ALIAS meta-model is provided by ASML. The ALIAS meta-model is part of the ASOME environment. Three important concepts can be found in the meta-models: 1) the Control System component, which describes what interfaces (specification models) are used by the component and which provides a realization (implementation models) of the component, 2) Control interfaces which describes what a component contains and how it is expected to behave and 3) a protocol, implemented as a state machine, which describes the SBS rules in the control models for both interface- and realization-models.

3.2.1 ALIAS models

Validating the transformation requires a test set. ASML currently uses a set of models defined in ASD which are being translated to ALIAS. The test set consists of 236 interface models and 208 design models. The difference between these numbers comes from 28 foreign components without design models.

Of these models, 108 design models and 201 interface models are compatible with the current version of ALIAS. These models are translated to ALIAS and used as an input to validate the transformation to mCRL2.

3.2.2 mCRL2 meta-model

The meta-model of mCRL2 is created by defining an Xtext grammar based on the grammar for the mCRL2 language. A meta-model is then generated from the Xtext grammar specification. This type of meta-model has various advantages and disadvantages:

Advantages

- The meta-model contains only the relevant mCRL2 features
- The meta-model is correct
- The Xtext grammar takes care of the model-to-text transformation
- Easy to implement/prototype

Disadvantages

- Syntax related meta-model rather than conceptual
- The grammar needs to be converted to LL(*) and cannot be used directly

For this project, a prototyping approach will be used. For this purpose, an easy to implement meta-model is favourable over constructing a feature-complete meta-model. The major drawback to this approach is the inability of Xtext to cope with parsing the mCRL2 grammar as it is not an LL(*) grammar. To solve this problem, additional keywords are introduced in the Xtext grammar which are afterwards removed to obtain mCRL2 syntax correct models along with rewriting the grammar and priorities of operators.

The Xtext grammar as used in the project can be found in Appendix A.

3.2.3 mCRL2 models

The meta-model of mCRL2 allows the creation of mCRL2 models in XMI format, the format used by EMF. With the Xtext text-to-model parsing, mCRL2 models may also be written in plain mCRL2 syntax according to the grammar. For our model-verification process, we generate mCRL2 models by transforming existing ALIAS model by applying a QVTo transformation on the ALIAS models, as shown in Figure **??**. These models are then translated to concrete mCRL2 syntax using the Xtext grammar.

3.3 Transformation

The transformation is implemented in QVTo. In Chapter **??** the formal specification of the translation is given. The created QVTo transformation is an implementation of this formal translation.

The input of a QVTo transformation is an ALIAS model and the output of a QVTo transformation is an mCRL2 model. Each ALIAS model can be transformed in three different ways: 1) the provided interface model, 2) the single-threaded realization and 3) the multi-threaded realization. Note that for models with a foreign component there is no realization and there are no models to translate for the latter two cases.

3.4 Validation

The validation of the translation is done in two ways. A manual validation of the state spaces produced by the transformation on small and/or well-known examples (e.g. the alarm system described in Chapter **??**). The second method of validating the translation is by transforming the set of given models to a set of mCRL2 models and generating state spaces using the mCRL2 tools. Each is explained more in the next sections.

3.4.1 Manual validation of state spaces

To gain confidence in the correctness of the generated state spaces, a manual inspection of the state spaces is performed. While checking the state spaces manually, only models which are expected to behave correctly and models which are expected to fail the verification by invoking an illegal trigger are considered. An overview of the semantic properties being manually validated is shown in Table **??**.

Property	Description
Illegal action	In models in which an illegal action is expected to occur, the model is manually checked for the occurrence of the illegal action.
Trace inspection	Verify that all expected traces are present in the labeled transition system
Trace equivalence	Running the ltscompare tool on the two resulting outputs from single-threaded and multi-threaded semantics gives insight in the relationship between single-threaded and multi-threaded models. Traces in the single-threaded output should be contained in the multi-threaded transformation.

Table 3.1: An overview of the properties manually being inspected.

3.4.2 Validation using a working set of models

To gain confidence in the coverage and scalability of the transformation, a working set of models is used. These models are transformed to mCRL2 and corresponding state spaces. The results of these transformations are gathered and converted into metrics for analysis. The process of obtaining the metrics required for the analysis is done by applying the model-to-model transformation to every interface and implementation model. The output of this step consists of mCRL2 specification files.

For every mCRL2 file, we apply the following shell script which transforms the mCRL2 files to linear process systems (LPS), linear process systems to labeled transitions systems and applies bisimulation reduction on the resulting state space. Information regarding the state spaces are stored in an output log.

```
echo ------------ $1 -----------------
mcrl22lps -bf --no-constelm -lregular2 $1 temp.lps &&
lpsconstelm -v temp.lps | lpsparelm -v > temp1.lps &&
lps2lts --todo-max=10000 -l1000000 -aqueueSizeViolation
    -t1 temp1.lps temp.aut -vrjittyc --cached &&
# ltsconvert -v -ebisim-gjkw temp.aut tempr.aut &&
# ltsinfo tempr.aut
echo =======================================
```

The output of this script is filtered for the number of states and transitions before and after applying bisumulation reduction. With these metrics, an exploration of the size of the state spaces is performed, as well as a figure on how many models pass transformation to mCRL2 and generate meaningful statespaces. The execution time of every tool is measured to test the scalability for larger ALIAS models. Models may time-out after a while if the resulting state space is too large to compute.

Chapter 4

Formal Transformation Specification

4.1 Formal Transformation Specification

The transformation of an ALIAS model to an mCRL2 specification is performed using a QVTO transformation. The transformation from ALIAS to mCRL2 remains confidential. In this chapter, we will discuss a translation from Mealy machines to mCRL2 instead. Mealy machines are based on state machines where each transition has an input and output. These properties are suffciently similar to ALIAS to provide a discussion on the translation to mCRL2 and sketch the concepts on how to translation a much more expressive language to mCRL2.

4.2 Mealy Machines

A Mealy machine is a finite-state machine. Unlike Moore machines, output values are determined both by the current state and input values and not just the current state. This behaviour allows us to create transition systems that rely on input, as if the Mealy machine is part of a controller.

4.2.1 Formal definition

A Mealy machine is a 5-tuple $(S, S_0, \Sigma, \Lambda, T)$, with the following elements:

- A set of states S. On S, the function labelling function label : $S \mapsto \mathbb{L}$ is defined, which maps each state $s \in S$ to a label.
- The initial state $S_0\ inS$.
- The input alphabet, Σ.
- The output alphabet, Λ.
- A set of transition functions $T : S \times \Sigma \mapsto S \times \Lambda$.

In order to perform the translation, we define the function $transitions : S \mapsto T$, which returns all the transitions belonging to a state $s \in S$.

4.2.2 Example Mealy machine

To illustrate the expressive power of a Mealy machine, we look at an example machine. Figure **??** shows a picture of a Mealy machine with three states 0, 1 and 2, where state 0 is the initial state. The input alphabet Σ is the set $\{0, 1\}$ and the output set Λ is $\{0, 1\}$.

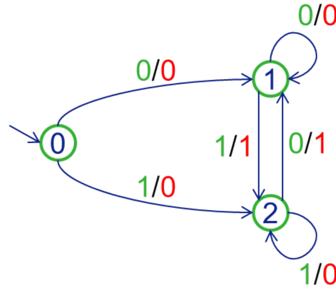

Figure 4.1: An example Mealy machine.

Fro mthe starting state, there are two transitions. The transition with input 0 goes to state 1, while the transition with input 1 goes to state 2. Both of these transitions have as output 0. From state 1, there is a self-loop if the next bit is a 0, and a transition to state 2 if the input is a 1. Only when the input is a 1 is the output also a 1. State 2 is similar as state 1, with a self-loop on the input 1 and a transition to state 1 on the input 0, in which case the output is a 1. The output of the Mealy machine can be described as if the value is 1 if and only if the current input bit differs from the previous input bit. The output is 0 otherwise.

4.3 Translation scheme

We want to translate the Mealy machine as described above to an mCRL2 specification. To do this, we consider a Mealy machine $M = (S, S_1, \Sigma, \Lambda, T)$, where $S = \{s_1, s_2, ..., s_n\}$, a set of n states. S_0, the initial state. The input alphabet is $\Sigma = \{x_1, x_2, ..., x_k\}$ and the output alphabet is $\Lambda = \{y_1, y_2, ..., y_m\}$, k and m are the sizes of the input and output alphabet. A transition t is of the form $t(s, i) = (s', o)$, where s is the current state and i the input on the state. s' is the output state and o the output.

The translation of the Mealy machine is given as follows:

Mealy (A) $\langle$

Sort Input struct = $|_{x:x\in\Sigma}$ [x] ;
Sort Output struct = $|_{y:y\in\Lambda}$ [y] ;

act
in : Input;
out : Output;

proc

$\forall_{s:s\in S}$ [
label(s) =
$+_{t:t\in transitions(s)}$ [in(i) . out(o) . s']
]

init S0;

$\rangle$

Symbols other than the ∀ symbol indicate that the elements in the translation are delimited with this symbol. The | in the sort "Input" thus means that every elements in the square brackets is separated with the | symbol.

In this translation of a Mealy machine, the input and output alphabets are translated to sort specification in mCRL2. The visible actions in the translation are the input action "in" and the output action "out", each carrying their respective alphabet in- and output.

4.3.1 Example translation

Figure **??** is used to show an example translation of a Mealy machine. The translation of the Mealy machine is given as follows:

```
Sort Input = struct 0 | 1;
Sort Output = struct 0 | 1;

act

in : Input;
out : Output;

proc
S0 = in(0) . out(0) . S1 + in(1) . out(0) . S2;
S1 = in(0) . out(0) . S1 + in(1) . out(1) . S2;
S2 = in(0) . out(1) . S1 + in(1) . out(0) . S2;

init S0;
```

The resulting translation matches the behaviour of the Mealy machine of Figure **??**. The initial state is S0, and from here we can do a 0/0 to state S1 or a 1/0 to state S2. Similar self loops are present in S1 and S2, respectivelly. The transition from S1 to S2 and S2 to S1 are also seen in the translation. Note that this translation is not valid in mCRL2, due to the names of the in- and output alphabets.

4.4 Composition of Mealy machines

We wish to decompose problems into smaller components. These components need to communicate with each other. Using Mealy machines, we can do this by defining a second Mealy machine whose input values overlap with input values from a component at a higher level. The output set of this higher level component may contain elements not in the input set of the lower level component (e.g. it controls two distinct Mealy machines each with a separate input alphabet). The input set of the lower level may contain more elements than the higher level component uses: transitions with these elements as input will simply not occur.

4.4.1 Example of a composition

Consider the Mealy machine from Figure **??**. We wish to add a component below this component, which shares the output alphabet and uses it as its own input alphabet. The output alphabet of this new Mealy machine is the set $\{yes, no\}$. This Mealy machine is shown in Figure **??**.

The output of this Mealy machine is *yes* once every three times the input is 1 and 0 otherwise. Whereas the output of the first Mealy machine is 1 after every change of bits, the output of the latter Mealy machine is *yes* after every three changes of bits and *no* otherwise when these two Mealy machines are placed in parallel and communicate with each other.

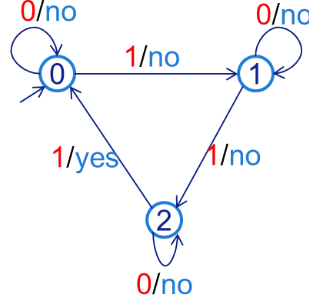

Figure 4.2: An example Mealy machine.

4.4.2 Translation scheme of compositional Mealy machines

In order to translate a compositional Mealy machine, we differentiate the models between an active client (e.g. the model from Figure **??**) and a service (e.g. the model from Figure **??**. A component consists of one client and zero or more services.

A client and server communicate with each other. In this communication, we denote the output of the client with "action", and the input of a server as an "event". An action and an event together are denoted as "operations". Input of clients and output of services remain unchanged.

Let M_C be a client Mealy machine and M_S be a set of service Mealy machines, each as described as in Section **??**. Then, the translation is given as follows:

MealyComposition (A) $\langle$

Sort Input struct $= |_{x:x\in\Sigma}$ [x_C] ;
Sort Operation struct $= |_{x:x\in\Sigma}$ [y_C] ;
Sort Output struct $= |_{y:y\in\Lambda}$ [y_M] ;

act
in : Input;
out : Output;
action, event, operation : Operation;

proc

$\forall_{s:s\in S_C}$ [
label(s) =
$+_{t:t\in transitions(s)}$ [in(i) . action(o) . s']
]

$\forall_{m:m\in M_S}$ [
$\forall_{s:s\in S_m}$ [
label(s) =
$+_{t:t\in transitions(s)}$ [event(i) . out(o) . s']
]
]

init
allow ({ in, out, operation },

comm ({ action|event→operation},
S0 || $||_{m:m \in M_S}$ [S_{m0}]
)) ;

⟩

In this translation, communication occurs between the client and service Mealy machines by means of the "action" and "event" actions. These events occur at the same time, as indicated by the communicate operator in the initial process.

4.4.3 Composition translation example

Consider again the two Mealy machines from Figures **??** and **??**. The first Mealy machine acts as a client, while the latter acts as a service. The composition of these two machines is translated to an mCRL2 specification. The mCRL2 specification is given as follows:

```
Sort Input = struct 0 | 1;
Sort Operation = 0 | 1;
Sort Output = struct no | yes;

act
in : Input;
out : Output;
action, event, operation : Operation;

proc
T0 = in(0) . action(0) . T1 + in(1) . action(0) . T2;
T1 = in(0) . action(0) . T1 + in(1) . action(1) . T2;
T2 = in(0) . action(1) . T1 + in(1) . action(0) . T2;

B0 = event(0) . out(no) . B0 + event (1) . out( no ) . B1;
B1 = event (0) . out(no) . B0 + event (1) . out( no ) . B2;
B2 = event (0) . out(no) . B0 + event (1) . out(yes) . B0;

init
allow( {in, out, operation},
comm ( {action—event-¿operation},
T0 —— B0 ));
```

In this specification, it can be seen that an action and event only occur at the same time, and this action is then called an operation. Due to the communication, the bottom level component only has an enabled action when the top level component performs the action "action".

Chapter 5

Validation

Confidence in the correctness of the transformation of ALIAS to mCRL2 is a requirement when defining the semantics: the corresponding state spaces need to adhere to the intended meaning of ALIAS . To this end, we validate the transformation in two different ways. First, we consider small, hand-crafted ALIAS models and manually inspect the results, by running the appropriate tools, to isolate features and by checking them independently. Secondly, we use an existing set of ALIAS models to validate the translation and establish the correctness of the translation itself.

5.1 Manual feature testing

A manual validation of the transformation is used to gain confidence in each individual feature present in ALIAS . There are five major features we wish to verify correctness for before moving to a working set of models. These features are a composition of used interfaces, the presence of the illegal action in an intentionally incorrect model, trace inclusion of the single-threaded statespace in the multi-threaded statespace, evaluation of expressions in guards and invariants and valued reply states.

For each of these features, we construct an ALIAS model and apply the transformation. Each of these models are kept as small as possible while retaining their meaning.

5.1.1 Composition of interfaces

The composition of a realization and its required interfaces is an important step in composing larger systems. For this validation we make use of the following features in ALIAS :

- Multiple required interfaces
- Actions: Invoke Operation, Send Notification, VoidReply
- Triggers: Operation Invoked, Notification Received, Internal Trigger
- Single-threaded execution model

This limited set of the expressive power of ALIAS is used to create a composition of various interfaces and a realization. First, a description of the model is given, followed by a discussion of the transformation of this model.

Model description

For this validation, we consider a model describing a simple alarm system. It makes use of two components: a sensor, which can be turned on and off, and while turned on can detect movement. The second component is a siren, which can be turned on and off.

When the alarm system itself is switched on, it will activate the sensor such that it is ready to detect movement. When the alarm system is switched on, two things may happen: either the alarm system is switched back off and the sensor will be turned off, or the sensor detects movement and sends a notification to the alarm system. The alarm system, when notified of the sensor being tripped will turn on the siren. The user of the alarm system can then turn off the alarm system again. The exact specification of the interface and its implementation are only available in the full report.

Validation

Figure ?? shows the state space of the described model. The initial state is marked with a green colour. Only the most important labels are shown on the transitions. Based on the state labels, it can be seen that after the alarm is switched on, the sensor is also being activated. After it is activated and replies are sent back, there is a point where two different actions are enabled: either the alarm system is switched off and we return, after a sequence of actions, back in the initial state. The other course of action involves the sensor detecting movement after an internal trigger occured: the alarm system is set off and the siren is turned on. Once the siren is turned on, the only allowed action is to disable the alarm system, which turns off the siren, deactives the sensor and eventually switches off the alarm system to return to the initial state.

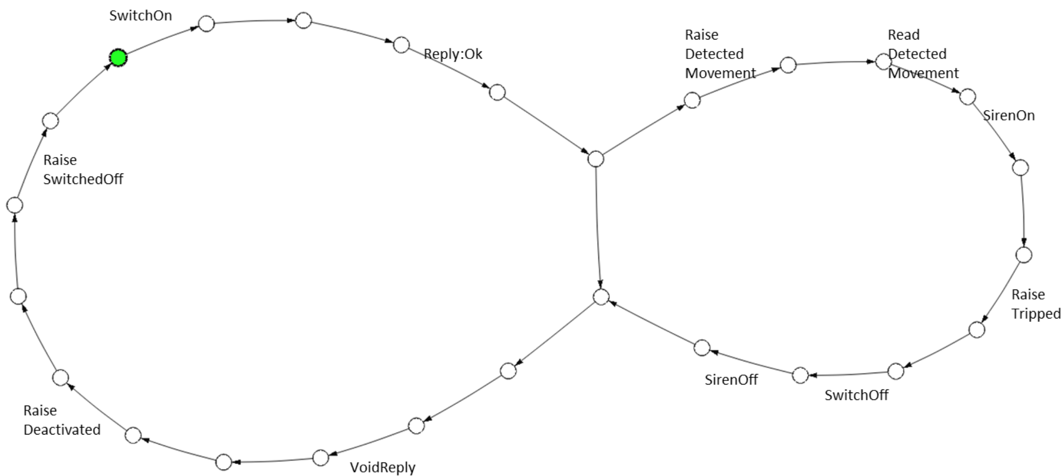

Figure 5.1: The state space of the single-threaded alarm system.

This small example model makes use of one implementation model, namely the component that defines the behaviour of the alarm system. It requires the use of two interfaces, the sensor and the siren. This small model shows that the transformation is capable of dealing with multiple required interfaces.

5.1.2 Illegal action

We show that the transformation produces an illegal action in the state space if we design a model where an illegal action is expected to occur. For this validation we use the multi-threaded execution model instead. The following features are used to validate this property of the transformation.

- Multiple required interfaces
- Actions: Invoke Operation, Send Notification, VoidReply

- Triggers: Operation Invoked, Notification Received, Internal Trigger
- Multi-threaded execution model

Model description

The same model description as for the composiition of interfaces is used. However, for this behaviour we consider the multi-threaded variant, instead. This results in a change of behaviour where both the deactivating and detected movement triggers occur at the same time. A race condition between these two triggers causes the alarm system to move into a state where it is not expecting to receive a detected movement notification while it is shutting down. This unexpected notification leads to an illegal transition.

Figure 5.2: The state space of the multi-threaded alarm system.

Validation

The state space of the multi-threaded alarm system is shown in Figure **??**. The action labels are hidden from this transition system. The same shape of the transition shape as the single-threaded variant can be found in the multi-threaded execution scheme. Actions are being interleaved as a result of the multi-threaded environment where each component can process events on their own.

As a result of the interleaving of actions, a sequence where both detected movement and switching off the alarm occur leads to an appendix that results in an eventual deadlock. The marked states with numbers 34, 35 and 39 are states reached after an illegal transition. The presence of these states indicate that a modelling error indeed lead to an illegal action.

5.1.3 Trace inclusion

The multi-threaded execution scheme allows more behaviour than the single-threaded execution model. However, every execution path in the single-threaded scheme is required to be included in the multi-threaded scheme. If the traces of the single-threaded scheme are included in the traces of the multi-threaded scheme, it is an indication of the correctness of the transformation.

Model description

The single-threaded and the multi-threaded variants of the models described above are re-used to verify this propery.

Validation

The tool ltscompare is used to validate this property. The followig command is used:

ltscompare alarmsystemST.lts alarmsystemMT
–equivalence=none –preorder=weak-trace –tau=internal,lockQ,unlockQ –verbose

Applying the tool on the single-threaded and multi-threaded transition systems of the alarm system, yields a result that the traces are weak trace equivalent and the inclusion is satisfied. Weak trace equivalence, rather than strong trace equivalence, is expected due to the naming differences between single-threaded and multi-threaded models, as well as the inclusion of additional queue handling actions for the single-threaded model.

5.1.4 Expressions

Expressions are used to control the flow of the specification. Expressions are used to evaluate guards for transitions and state invariants which impose restrictions on the states. Evaluation of guards and state invariants, as well as updating the values of state variables are important features used to control the sequence of transitions. In this verification step, a model making full use of state invariants and guards are used. The state invariant will be set up to cause a violation at run-time to verify the state invariant feature is transformed as expected. The invariatant action does not cause a dead-lock in the implementation: the action is enabled, but does not need to be executed. Due to this behaviour of the transformation, an integer range violated is also being checked with the same model. The model uses the following features in ALIAS :

- State variables
- State variable updates
- Range violations
- State invariants
- Guards
- Single-threaded execution model

Model description

The model consists of a single state, with a single trigger: a valued reply operation. There are two transitions in the model and two state variables, one of which is a counter and the other one is used to generate an error. The range of the second variable is limited to [0,2]. One transition increments the counter, the second transition decrements the counter. The second transition which decrements the counter also increments a state variable called 'error'. The increment and

decrement counters are guarded with expressions such that they are only enabled one at a time, and taking turns. The state invariant states that the value of 'error' is always smaller than two.

The model intentionally violates the invariant after performing the decrement transition twice: error is set to two and the state invariant is violated. The state space generation is not stopped in this case, and two transitions later the range of error is violated, and a range violation action is also visible in the state space. While generating the state space of a model, once an illegal action, range violation or invariant violation occurs, the state space generation stops and a trace is procuded instead. For this validation purpose however, the whole state space is generated to visualize the actions.

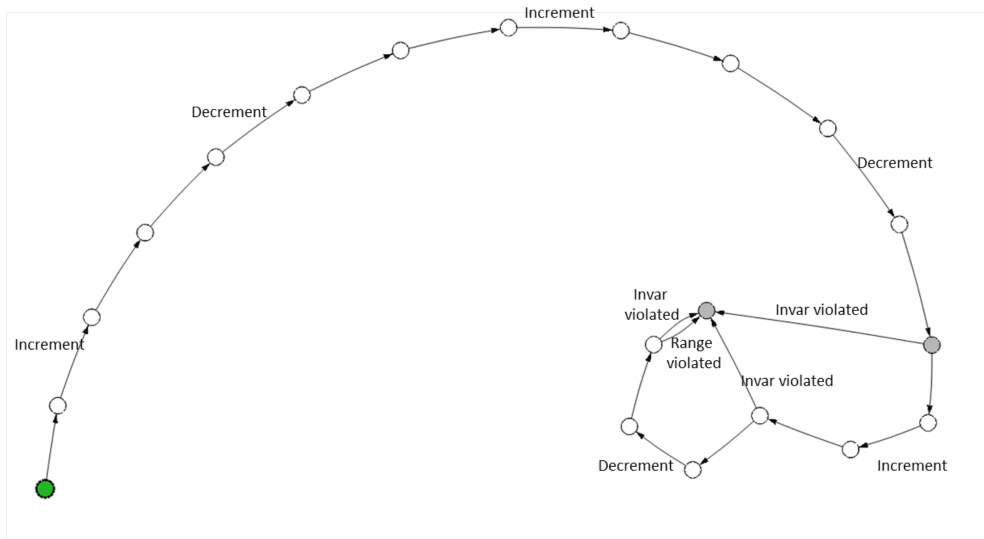

Figure 5.3: The state space of an example model validating the usage of expressions.

Validation

Figure ?? shows the transition system from the described model. A sequence of increment and decrement transitions eventually leads to a state where the invariant is violated. At this point, the action invar_violated is enabled to be taken. The state space generation continues until the range violated occurs when the error variable takes on the value 3 and exceeds its declared range, at which point there is no suitable transition available and the state space generation is stopped.

5.1.5 Valued Reply states

To make decisions based on the return value of an invoked operation, there is a special type of states called Valued Reply States. In this section, a small model is used to test whether the returned value of a method call on a valued reply generator is the expected responce. This verification makes use of the following features of ALIAS :

- Action: Invoke Valued Reply Operation
- Trigger: Valued Reply Operation Invoked, empty trigger
- Guards
- Single-threaded execution model

Model description

The model consists of two parts: a valued reply generator interface, which replies with one of the following values: Ok, NOk or Failed. The other part is a realization which invokes the valued reply generator and receives a value. Based on the result of the value, the realization replies with the value Yes, No or Failed according to the received value from the generator.

Validation

In Figure **??** the transition system of the model is shown. There are three sequences before returning back to the intial state, one for each of the possible valued replies. For each branch, the valued replies cause the appropiate further actions: Ok is followed up by Yes, NOk is followed up by No and Failed is followed up by Failed. This is the behaviour that is expected from the model.

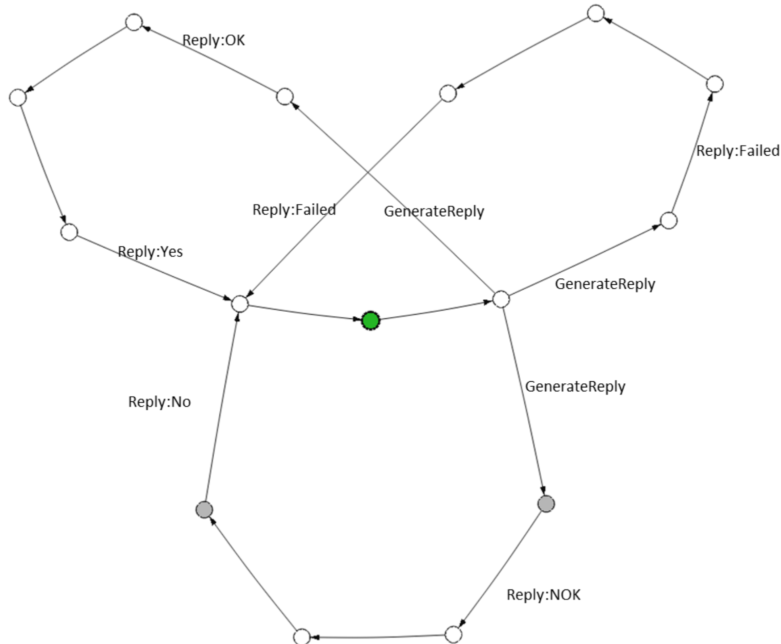

Figure 5.4: The state space of an example model validating the usage of valued replies.

5.2 ALIAS set of models

We have access to a set of 126 models which are used in production and verified to be correct using legacy tools. All models are transformed to mCRL2 and we collect the following information of every transformation:

- A status message: Ok, Failed, TimeOut
- The number of states
- The number of states after bisumulation reduction
- The number of transitions
- The number of transitions after bisimulation reduction

Execution model	ALIAS	mCRL2	lps	lts
Single-threaded	108	108	108	108
Multi-threaded	108	108	108	42

Table 5.1: The table shows the number of models that are generated during each transformation step. Only the implementation models are considered.

Every realization model is transformed according to both of its execution schemes, namely single-threaded and multi-threaded. Realization models implement interfaces, which are translated along with the realization. We present the results of the realization models. All interface models are found to be translated to valid mCRL2 specifications as well as valid state spaces. The transformation results for realization models is displayed in Table **??**.

The table shows that all the ALIAS models can be translated to corresponding mCRL2 models. Out of the 108 models, 108 models can be transformed to a LPS (Linear Process System). All 108 LPS files can be generated to state spaces under single threaded execution, but only 42 can be generated for multi-threaded models. This is because the test set is intended for single-threaded usage, and 66 models were found to not be compatible with multi-threaded semantics.

Chapter 6

Conclusions

In this work we defined the formal semantics of ALIAS using mCRL2. This work resulted in the definition of various aspects of the semantics of the language involving single- and multi-threaded execution models, reply values and expressions. This led to a prototype being built to transform ALIAS models into mCRL2 models.

The defined semantics of the execution models are based on a queue containing messages from used services of a component. The semantics and purpose of the queue has a slightly different meaning in either of the two execution semantics. In the single-threaded execution semantics, the queue acts both as the receiver and controller of the queue as well as the element that controls the execution flow. A queue that serves as a lock on the component that prevents the component from starting more paths of execution was found to be an effective method to direct the flow of control. This matches the expected behaviour of the single-threaded execution model.

For the multi-threaded variant, this locking behaviour is disabled. This allows each component to act independently and run in parallel. This method was found to be suitable as a means to simulate a model as if it were multi-threaded.

Next, a prototype translation was made using a model-to-model transformation. The resulting mCRL2 model is then transformed to mCRL2 syntax and used to create a linear process system and finally a labeled transition system. ALIAS models are based on state machines. Every specification and implementation model is translated to mCRL2 process equations, where every state corresponds with a state in the model and every transition between states corresponds with a summand in the process equation.

This breakdown of components into smaller pieces is found to be practical: as a result of the breakdown in states and transitions, the handling of state variables and parameters is kept simple and only need to be passed on to the next process equation.

The generated state spaces are used to validate the correctness of the transformation. We found that, for small cases, the transformation to mCRL2 is working as intended. The studied cases involve the most important cases supported by ALIAS , namely composition of interfaces, single- and multi-threaded execution, illegal behaviour, expressions and valued replies.

To validate the translation to a bigger set of models, ASML provided a set of 126 models used in production. All of those models could be translated to mCRL2 models. In subsequent steps, some models failed due to timeouts due to timing constraints. This could mean that either these models are too large to be verified, or that there is a fault in the translation causing the state space to explode. The cause of this is not explored as of yet.

Future work

The current work is based on an Xtext generated mCRL2 meta-model. This leads to a number of inefficiencies compared to a meta-model that is fully functional and has a better structure. An improved meta-model will allow for a cleaner and more maintainable implementation of the translation.

Not every feature in ALIAS is fully translated to mCRL2. In regards to storage parameters, there is are some parts semantics that need to be investigated further and discussed to formulate how these storage parameters are to be used and their correct usage is still to be verified.

Bibliography

[1] Ju An Wang, Towards component-based software engineering. Journal of Computing Sciences in Colleges Volume 16 Issue 1, Oct 2000 Pages 177-189

[2] Mealy, Georgy H., A Method for Synthesizing Sequential Circuits. BSTJ 34: 5. September 1955

[3] ASD:Suite User Documentation. http://community.verum.com/documentation.aspx

[4] J.F. Groote, M. R. Mousavi, Modeling and Analysis of Communicating Systems, August 2014

[5] Thomas Kühne, Matters of (Meta-) Modeling. Software and Systems Modeling, 5(4), 2006, pp. 369-385.

[6] QVTo examples and information. https://wiki.eclipse.org/QVTo

[7] Gehan M. K. Selim, Shige Wang, James R. Cordy, Juergen Dingel, Model Transformations for Migrating Legacy Models: An Industrial Case Study, Volume 7349, Lecture Notes in Computer Science, pp 90-101

[8] Xtext documentation and tutorials. http://www.eclipse.org/Xtext/index.html

Appendix A

Xtext grammar

The Xtext grammar as is used to generate the metamodel.

```
grammar org.xtext.mcrl2.start.MCRL2 with org.eclipse.xtext.common.Terminals

generate mCRL2 "http://www.xtext.org/mcrl2/MCRL2"

mCRL2Spec
    : elements+=mCRL2SpecElt* initial=Init elements+=mCRL2SpecElt*
;

mCRL2SpecElt
    : SortSpec
    | ActSpec
    | ProcSpec
    ;

Init
    : 'init' initial=ProcExpr ';'
    ;

ProcExpr
    : ProcExprC
    ;

ProcExprC returns ProcExpr
    : ProcExprD ({ChoiceExpression.left=current} '+' right=ProcExprD)*
    ;

ProcExprD returns ProcExpr
    : ('dist' vars=VarsDeclList '[' de=DataExpr ']' '.' ) pe=ProcExprSum
    | ProcExprSum
    ;

ProcExprSum returns ProcExpr
    : 'sum' sum=SumExpression
    | ProcExprP
    ;

SumExpression
```

```
    : vars=VarsDeclList '.' pe=ProcExprP
    ;

ProcExprP returns ProcExpr
    : ProcExprL ( {ParallelExpression.left=current} '||' right=ProcExprL)*
    ;

ProcExprL returns ProcExpr
    : ProcExprITE ({LeftAssocExpression.left=current} '||_' right=ProcExprITE)*
    ;

ProcExprITE returns ProcExpr
    : ProcExprU
    | 'ifs_' dexu=DataExprUnit '->' ctrue=ProcExprU ('<>' cfalse=ProcExprU)?
    ;

ProcExprU returns ProcExpr
    : ProcExprSeq ({UntilExpression.left=current} '<<' right=ProcExprSeq)*
    ;

ProcExprSeq returns ProcExpr
    : ProcExprAt ({SeqExpression.left=current} '.' right=ProcExprAt)*
    ;

ProcExprAt returns ProcExpr
    : ProcExprCM ('@' deu+=DataExprUnit)?
    ;

ProcExprCM returns ProcExpr
    : ProcExprTerminal ({CommMergeExpression.left=current} '|' right=ProcExprTerminal)*
    ;

ProcExprTerminal returns ProcExpr
    : 'act_' atomicAction=[Action] ( '(' del=DataExprList ')' )?
    | 'proc_' atomicProcess=[Process] ( '(' del=DataExprList ')' )?
    | atomicDelta = 'delta'
    | atomicTau = 'tau'
    | 'block' '(' blockset=ActIdSet ',' pe=ProcExpr ')'
    | 'allow'  '(' allowList=MultiActIdList ',' pe=ProcExpr ')'
    | 'hide' '(' hideset=ActIdSet ',' pe=ProcExpr ')'
    | 'rename' '(' renameList=RenExprList ',' pe=ProcExpr ')'
    | 'comm' '(' commList=CommExprList ',' pe=ProcExpr ')'
    | '(' ProcExpr ')'
    ;

ActIdSet
    : '{' ActionList '}'
    ;


MultiActIdList
    : '{' multiactions+=MultiActId ( ',' multiactions+=MultiActId)* '}'
    ;
```

```
MultiActId
    : acts+=[Action] ( '|' acts+=[Action])*
    ;

RenExprList
    : '{' renames+=RenExpr ( ',' renames+=RenExpr)* '}'
    ;


RenExpr
    : act=[Action] '->' renact=[Action]
    ;


CommExprList
    : '{' communications+=CommExpr ( ',' communications+=CommExpr)* '}'
    ;

CommExpr
    :  acts += [Action] ( '|' acts += [Action] )? '->' toact=[Action]
    ;

VarsUseList
    :  variables+=[Variable] ( ',' variables+=[Variable])*
    ;

VarList
    : vars+=Variable ( ',' vars+=Variable)*
    ;

Variable
    : name=ID
    ;

VarsDeclList
    : variables+=VarsDecl ( ',' variables+=VarsDecl )*
    ;

VarsDecl
    : vl=VarList ':' se=SortExpr
    ;


SortExpr
    : SortExprTerminal
    ;

SortExprTerminal returns SortExpr
    : sort=[Sort]
    | int='Int'
    | pos='Pos'
    | nat='Nat'
    | bool='Bool'
    | list='List' '(' se=SortExpr ')'
```

```
    | set='Set' '(' se=SortExpr ')'
    | bag='Bag' '(' se=SortExpr ')'
    | fset='FSet' '(' se=SortExpr ')'
    | fbag='FBag' '(' se=SortExpr ')'
    | sortb='sortb' '(' se=SortExpr ')'
    | 'struct' ConstrDeclList
    ;

ConstrDeclList
    : constructors+=Struct ( '|' constructors+=Struct )*
    ;


DataExprUnit
    : 'dataterm_' '(' dataterm=DataExpr ')'
    | 'datalist_' '(' datalist+=DataExprList ')'
    ;

DataExprUnitTerminal returns DataExprUnit
    : identifier=ID
    | int=INT
    | bool='true'
    | bool='false'
    | not='!' dexu=DataExprUnit
    | minus='-' dexu=DataExprUnit
    | size='#' dexu=DataExprUnit
    | expr=DataExpr
    ;

DataExpr
    : DataExprImpl
    ;


DataExprImpl returns DataExpr
    : DataExprConj =>({ImplicationExpression.left=current} '=>' right=DataExprConj)*
    ;

DataExprConj returns DataExpr
    : DataExprDisj =>({ConjunctionExpression.left=current} '&&' right=DataExprDisj)*
    ;

DataExprDisj returns DataExpr
    : DataExprEq =>({DisjunctionExpression.left=current} '||' right=DataExprEq)*
    ;

DataExprEq returns DataExpr
    : DataExprIEq =>({EqualityExpression.left=current} '==' right=DataExprIEq)*
    ;

DataExprIEq returns DataExpr
    : DataExprLT =>({InequalityExpression.left=current} '!=' right=DataExprLT)*
    ;
```

```
DataExprLT returns DataExpr
    : DataExprLE =>({LessThanExpression.left=current} '<' right=DataExprLE)*
    ;

DataExprLE returns DataExpr
    : DataExprGE =>({LessEqualExpression.left=current} '<=' right=DataExprGE)*
    ;

DataExprGE returns DataExpr
    : DataExprGT =>({GreaterEqualExpression.left=current} '>=' right=DataExprGT)*
    ;

DataExprGT returns DataExpr
    : DataExprIn =>({GreaterThanExpression.left=current} '>' right=DataExprIn)*
    ;

DataExprIn returns DataExpr
    : DataExprLC =>({InExpression.left=current} 'in' right=DataExprLC)*
    ;

DataExprLC returns DataExpr
    : DataExprLS =>({ListConsExpression.left=current} '|>' right=DataExprLS)*
    ;

DataExprLS returns DataExpr
    : DataExprConc =>({ListSnocExpression.left=current} '<|' right=DataExprConc)*
    ;

DataExprConc returns DataExpr
    : DataExprAdd =>({ConcatenateExpression.left=current} '++' right=DataExprAdd)*
    ;

DataExprAdd returns DataExpr
    : DataExprSub =>({AdditionExpression.left=current} '+' right=DataExprSub)*
    ;

DataExprSub returns DataExpr
    : DataExprDiv =>({SubtractionExpression.left=current} '-' right=DataExprDiv)*
    ;

DataExprDiv returns DataExpr
    : DataExprIDiv =>({DivisionExpression.left=current} '/' right=DataExprIDiv)*
    ;

DataExprIDiv returns DataExpr
    : DataExprMod =>({IDivisionExpression.left=current} 'div' right=DataExprMod)*
    ;

DataExprMod returns DataExpr
    : DataExprMult =>({ModulusExpression.left=current} 'mod' right=DataExprMult)*
    ;

DataExprMult returns DataExpr
    : DataExprAt =>({MultiplicationExpression.left=current} '*' right=DataExprAt)*
```

```
    ;

DataExprAt returns DataExpr
    : DataExprWhere =>({ElementAtExpression.left=current} '.' right=DataExprWhere)*
    ;


DataExprWhere returns DataExpr
    : DataExprFU //('whr' assignments+=AssignmentList 'end')?
    ;

Assignment returns DataExpr
    : ID '=' DataExpr
    ;

AssignmentList returns DataExpr
    : assignments+=Assignment ( ',' assignments+=Assignment)*
    ;

DataExprFU returns DataExpr
    : DataExprFA //('[' d1=DataExpr '->' d2=DataExpr ']')?
    ;

DataExprFA returns DataExpr
    : DataExprTerminal //('(' dlist+=DataExprList ')')?
    ;


DataExprTerminal returns DataExpr
    : 'str_' struct=[Struct]
    | 'tail' '(' tail=DataExpr ')'
    | 'head' '(' head=DataExpr ')'
    | 'dele_' delegate=DataExprUnit
    | identifier=[Variable] ('=' datax=DataExpr)?
    | value=INT
    | bool='true'
    | bool='false'
    | curly='{' (':')? '}'
    | square= '[' ']'
    | '[' DataExprList ']'
    | '{' vd=VarDecl '|' de=DataExpr '}'
    | '{' DataExprList '}'
    | '(' DataExpr ')'
    | nott='!' '(' de=DataExpr ')'
    | minus='-' de=DataExpr
    | size='#' '(' de=DataExpr ')'
    | forall='forall' vdl=VarsDeclList '.' de=DataExpr
    | exists='exists' vdl=VarsDeclList '.' de=DataExpr
    | lambda='lambda' vdl=VarsDeclList '.' de=DataExpr
    ;

DataExprList
    : data+= DataExpr ( ',' data+=DataExpr )*
    ;
```

```
VarDecl
    : vrb=Variable ':' se=[Sort]
    ;



ActSpec
    : 'act' acts+=(ActDecl)+
    ;

ActDecl
    : ActionList ( ':' sorts+=SortProduct )?
    ';'
    ;

SortProduct
    : sorts+=SortTerminal ( '#' sorts+=SortTerminal )*
    ;

SortTerminal
    : sort = [Sort]
    | int = 'Int'
    ;

ProcSpec
    : 'proc' processes+=(ProcDecl)+
    ;

ProcDecl
    : proc=Process ( '(' vdl=VarsDeclList ')' )?
    '=' process=ProcExpr ';'
    ;

SortSpec
    : 'sort' sorts+=(SortDecl)+
    ;

SortDecl
    : sorts+=SortList ';'
    | sort=Sort '=' se=SortExpr ';'
    ;


IdsDecl
    : IdList ':' SortExpr ';'
    ;


Action
    : name=ID ( '(' del=DataExprList ')' )?
    ;

Process
```

```
    : name=ID
    ;

Sort
    : name=ID
    ;

Struct
    : name=ID
    ;


ActionList
    : actions+=Action ( ',' actions+=Action)*
    ;


SortList
    : sorts+=Sort ( ',' sorts+=Sort)*
    ;

IdList
    : ID ( ',' ID)*
    ;
```