Bounded Model Checking for RSL using RT-Tester

Peter Holm Østergaard

Kongens Lyngby 2016

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The goal of this thesis is to implement a translator that takes the specification of a given RSL state transition system, and translates it into a corresponding model in RT-Tester. The purpose of this translation is to combine the powerful specification capabilities of RSL with the model checking capabilities of RT-Tester.

The method chosen to accomplish this goal, is to design an intermediate language which acts as a concise, concrete syntax for RT-Tester. RSL specifications are translated to this intermediate language, which is then parsed into a model in RT-Tester to be model checked.

The subset of RSL, which the translator accepts, is presented, and the syntax of the intermediate language is defined in the form of a BNF grammar. The translation to the intermediate language is defined mathematically using a system of inference rules.

The implementation of the translator and parser is described, and the implementation is tested to show that it matches the design.

In conclusion the goal of the thesis is deemed fulfilled, albeit only for a subset of RSL.

Summary (Danish)

Målet for denne afhandling er at implementere en oversætter der tager specifikationen af et givet RSL transitionssystem, og oversætter det til en tilsvarende model i RT-Tester. Formålet med denne oversættelse er, at kombinere RSLs stærke specifikationsevner med RT-Testers evne til at modeltjekke.

Den valgte metode til at opnå dette mål, er at designe et mellemliggende sprog der kan agere som en kortfattet, konkret syntaks for RT-Tester. RSL specifikationer bliver så først oversat til dette mellemliggende sprog, som så derefter parses til en model i RT-Tester og bliver modeltjekket.

Den delmængde af RSL som oversætteren accepterer bliver præsenteret, og syntaksen af det mellemliggende sprog bliver defineret i form af en BNF grammatik. Oversættelsen til det mellemliggende sprog bliver defineret matematisk ved hjælp af et system af inferensregler.

Implementeringen af oversætteren og parseren bliver beskrevet, og implementeringen bliver testet for at vise at den er i overensstemmelse med designet.

Afslutningsvis bliver målet med afhandlingen anset som værende opnået, dog kun for en delmængde af RSL.

Preface

This thesis was prepared at DTU Compute, Technical University of Denmark, in fulfilment of the requirements for acquiring an M.Sc. in Engineering, and is credited with 30 ECTS points. It was prepared in the period of 24th of August 2015 to 25th of January 2016.

The thesis deals with bounded model checking of RSL state transition systems, by way of translating RSL specifications into models in RT-Tester.

The thesis consists of this report along with an associated file *thesis.zip*, containing the software which has been developed during the thesis work.

The thesis has been supervised by Associate Professor Anne Elisabeth Haxthausen.

Lyngby, 25-January-2016

Peter Holm Østergaard

Peter Holm Østergaard

Acknowledgements

First and foremost, I would like to thank my supervisor Anne Elisabeth Haxthausen for her help and support throughout this project. Our weekly meetings, in particular, have been an invaluable source of motivation and inspiration.

I would also like to thank Linh Hong Vu for his help with the technical side of the thesis work, and in particular his help with understanding and using RT-Tester. Without his help, I would never have been able to complete the RT-Tester part of the implementation.

Finally, I would like to thank Jan Peleska for granting me permission to access the code base of RT-Tester, without which this project could not exist.

Contents

Chapter 1

Introduction

In this chapter, the goal of this thesis is presented, followed by the motivation for fulfilling this goal. Furthermore, the prerequisites necessary to fully understand this thesis is listed, as well as a overview of the contents of this report.

1.1 Goal

The goal of this thesis is to implement a translator that takes the specification of a given RSL state transition system, and translates it into a corresponding model in RT-Tester. The purpose of this translation is to combine the powerful specification capabilities of RSL with the model checking capabilities of RT-Tester.

1.2 Motivation

Testing the correctness of software is a major part of software engineering. It is estimated that 30-50% of the cost of software projects are spent on testing

[7]. And even then, bugs and malfunctions can still occur, which is especially problematic in safety-critical systems.

Model checking offers a way to mathematically prove whether a system meets the required specification, and it could therefore be viewed as a solution the current issues with manual testing. However, due to the state explosion problem, it is only feasible to model check systems with a relatively small number of components. Although, this boundary continues to be pushed as model checking techniques are improved, and more powerful computers are developed.

RT-Tester is a test automation tool, which can also be used to perform bounded model checking. The tool has recently been used push the boundary of model checking interlocking railway systems, in terms of the size of the systems being verified [8]. However, when feeding the models being verified to RT-Tester, it is currently only possible using SyML/UML [6].

RSL is specification language designed for specifying models mathematically, and is therefore much better suited as an input language for RT-Tester, compared to SysML/UML.

1.3 Reader prerequisites

In formulating this thesis, it is assumed that the reader has a basic understanding of the following topics:

- Model checking
- Translation concepts, such as lexers, parsers and syntax trees
- The RAISE specification language (RSL)
- Inference rules

1.4 Chapter overview

This thesis contains the following chapters:

Chapter 2 - Project Context The relevant context and background is presented, such as how model checking, RSL and RT-Tester fit together in this project. The existing work related to this project is also discussed.

Chapter 3 - Method Analysis The overall method of the project is analysis, and the chosen method is presented.

Chapter 4 - Design Choices and Analysis The translatable subset of RSL is presented, and the design of the translation is analysed and presented in the form of a system of inference rules.

Chapter 5 - Implementation The implementation of the RSL translation is outlined, as well as the implementation of the intermediate language parser and model checking in RT-Tester.

Chapter 6 - Testing the RSL translator The implementation is tested in terms of whether the translator behaves as specified in Chapter 4.

Chapter 7 - Demonstration of software application The application of this project is demonstrated by showing the translation and model checking of an RSL specification.

Chapter 8 - User Guide A user guide of how the user can build and execute the program is presented.

Chapter 9 - Further Development The more interesting areas of this project, which could be further developed in the future, is discussed.

Chapter 10 - Conclusion A conclusion of the project is presented and the goal and content of the thesis is reflected on.

CHAPTER 2

Project context

In this chapter, the relevant context of this project will be presented.
The following topics form the main content of this chapter:

- An outline of how model checking works.
- A description of RSL, and how it relates to this project.
- A description of RT-Tester, how it relates to this project, and how to use it for model checking purposes.
- A discussion on previous work that relates to this project, and what has been accomplished.

2.1 Model checking

Model checking is a process of system verification. In short, the process consist of three steps:

- Creating a model, which is a mathematically precise and unambiguous representation of the system.

- Specifying the system properties which are to be verified.
- Feeding the model and properties to a model checking tool, where all the possible states of the model are explored and checked against the given properties.

These steps are also illustrated in Figure 2.1.

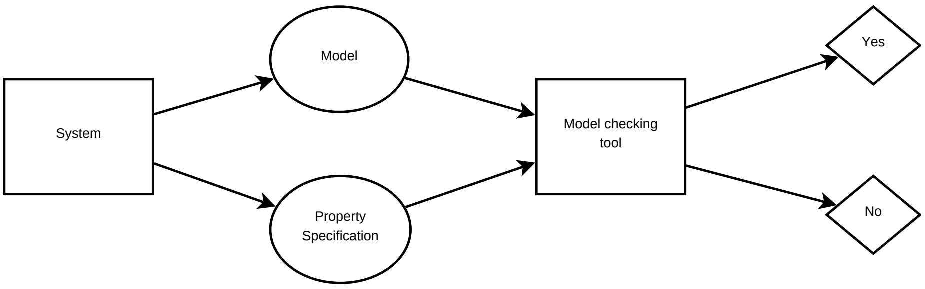

Figure 2.1: The process of model checking

The first step, where the model and properties are created, is a very important part of the verification process. If the model is not a completely accurate representation of the system, any verification result using the faulty model cannot be trusted [7].
However, this project does not consider the creation of the model and properties, and whether model accurately represent any system. Instead the focus is entirely on the final step, where a given model and property specification is given to a model checking tool.

2.2 RSL

RSL (RAISE specification language)[1] was developed as part of RAISE (Rigorous Approach to Industrial Software Engineering), which is a product consisting of three parts [10]:

- A software development method
- A formal specification language (RSL)
- Tools supporting the language and method

The focus of RAISE is on the use of *formal methods*, which is a mathematically based technique for specifying, developing and verifying both software and hardware systems. The motivation for using formal methods in general, is the increased reliability and robustness it offers, compared to conventional methods.

Because RSL was developed with formal methods in mind, it is a powerful language for making system specifications. It is a formal language, and it has features from many different language styles [11], such as:

- Property-oriented and model-oriented styles
- Applicative and imperative styles
- Sequentiality and concurrency styles

The primary advantages of using formal specifications such as RSL, is that formal specifications are precise and easier allows for mathematical analysis [9]. With the tools currently available for RSL, is not possible to execute or verify the specification directly, without translating it to some other representation. This is where RT-Tester comes in.

2.3 RT-Tester

RT-Tester is a commercial product, developed by Verified Systems International GmbH [6]. It is a test automation tool for automatically generating, executing and evaluating tests. RT-Tester is written in C++.

For this project, the main feature of interest in RT-Tester, is the ability to verify properties defined in linear temporal logic (LTL) on a given model specification. RT-Tester has its own way of representing the models internally, using a number of C++ classes and objects. So part of the challenge of the project is to find a way to translate a given specification (in this case an RSL specification) into RT-Testers internal model representation.

2.3.1 Model checking using RT-Tester

When performing model checking using RT-Tester, there are a few things to consider.
The model checking in RT-Tester is done using a SMT (Satisfiability Modulo Theory) solver, which is similar to a SAT (Boolean Satisfiability Problem) solver, except SMT solvers can include other theories, such as theories for integers, real numbers, arrays and other data structures.

When given a model and a LTL formula, the SMT solver works by looking for an interpretation of the model which satisfies the formula. In other words, it looks for witnesses to a given LTL formula. This means that the solver does not check whether the given LTL formula is satisfied in all possible computations of the model, but rather attempts to find a single computation which is satisfied. This needs to be taken into account when the user constructs the LTL formulae that needs checking.
For instance, consider the following LTL formula, checking whether it holds globally that no errors occur:

$$G(\neg error)$$

If this formula is checked by the solver, it will attempt to find a single computation without errors, and will say nothing about possible errors in other computations.
The way to handle this, is by using the concept of proof by contradiction. We instead negate the formula, and model check the following:

$$F(error)$$

If the solver now is unable to find a witness for this formula, it must hold that no error occurs in any of the possible computations. A similar case, is when checking that a certain state is eventually reached:

$$F(success)$$

This can also only be checked by using proof by contradiction, and instead looking for the absence of witnesses satisfying the following formula:

$$G(\neg success)$$

Another thing to consider, is that the SMT solver only performs *bounded* model checking, where a model is only being explored for a finite number of computational steps. Therefore the verification only guarantees the result up to a certain point.
It would be much preferred if the model checking using RT-Tester could provide assurances regardless of the number of transitions taken. Although it has not been done for this project, it is possible to achieve this by combining bounded model checking with *k-induction* [5].
The general idea of this method is to prove that a property $G(\phi)$ holds in all possible steps, if the following holds for $k > 0$:

The base case: ϕ holds for any computational path of length k, starting from the initial state.

The induction step: If ϕ holds for any computational path of length $k+1$ starting from an arbitrary reachable state, then it also holds for any next state after the initial.

Implementing an automated way of doing k-induction would be an interesting topic for future work.

2.4 Related work

The most closely related work, is the Ph.d. thesis *Formal Development and Verification of Railway Control Systems* [8] by Linh Hong Vu. In this thesis, RT-Tester is used along with k-induction to perform verification of railway control systems. The models being verified are specified using a domain specific language, which is inspired by RSL and even reuses subsets of RSL.
The main difference in relation to this project is that this project attempts to translate any RSL specification, rather than railway systems specified for a certain domain.

SAL (Symbolic Analysis Laboratory) is a specification language which is supported by a tool suite that includes a number of different model checkers. As part of the rsltc tool, which was developed to support the RAISE specification language, there exists a translator from RSL to SAL, which enables the model checking of RSL specifications using that translator [3].
One thing to note here, is that SAL, like RSL, is a specification language. It is therefore conceptually more similar to RSL compared to the internal representation of models in RT-Tester. In general, specification languages tend to offer a high level of abstraction, whereas RT-Tester is represented at a lower level of abstraction, offering little in terms of data structures.

In the master thesis *Model Checking RAISE Specifications using nuXmv* [13] by Kim Sørensen, a translator from RSL specifications to nuXmv is developed. nuXmv is a symbolic model checker [14], so there is a strong resemblance between that project and this one, with the only difference being the target model checker.

There also exists a number of other RSL translators, namely translators to SML, C++ and PVS [2]. While the target language in these translations are not very relevant to this project, the way in which the RSL specifications are processed, before the target language is generated, may be similar. The C++ translator in particular may be relevant, seeing as RT-Tester is implemented in C++, though the translation in this project is to an existing C++ system rather than to the

C++ language itself.

Chapter 3

Method analysis

In this chapter, the overall translation method will be analysed.
The following topics form the main content of this chapter:

- A very brief outline of the steps involved in language translation.
- A method analysis of how RSL can be translated to RT-Tester.
- A discussion of existing tools, which could be used in the implementation of the translator.
- A presentation and justification of the chosen method of translation.

A discussion of the more low-level design choices relevant to the translation, can be found in Chapter 4.

3.1 Language translation in general

Translating programming languages is a common problem in computer science, and as such there already exists a lot of theory developed on the subject.
In short, a typical language translation is done using the following steps:

- Lexing, where characters from the source language are grouped into tokens.
- Parsing, where the tokens are grouped into syntactical units, forming a parse tree or abstract syntax tree.
- Code generation, where the parse tree or abstract syntax tree is transformed into code for the target language.

These steps can be accomplished in a variety of different ways, but since language translation is a common problem, a lot of tools already exist to help facilitate these steps, some of which will be presented in Section 3.3.

3.2 Translation methods

It is important to note, that RT-Tester is a program implemented in C++, rather than being an actual language in and of itself. Because of this, when an RSL specification is translated to RT-Tester, the resulting code is primarily a series of C++ object constructions. Compared to the original RSL specification, this is neither very concise nor readable for the user. Because of this, it is very difficult and time consuming to manually create models in RT-Tester. Ideally, this problem of conciseness and readability should be addressed by the translation method.

3.2.1 Direct translation

The most intuitive method of translating an RSL specification to the corresponding model in RT-Tester, is to do it in a single step. That is, the RT-Tester model is generated directly based on the RSL specification, as seen in Figure 3.1. The advantage of doing the translation directly, is that the process becomes conceptually simpler. However, this approach does nothing to handle the problem outlined in the paragraph above.

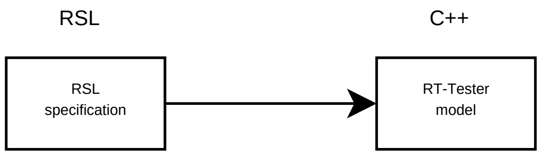

Figure 3.1: Translation method 1 - Direct translation

3.2.2 Translation using an intermediate step

Another method could be to add another step to the translation, such that the RSL specification is first translated to some intermediate language representation, which is then translated into an RT-Tester model. This approach is visualized in Figure 3.2.

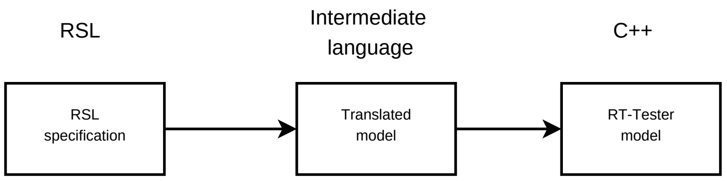

Figure 3.2: Translation method 2 - Intermediate step

This extra step can have a number of advantages, depending on how the intermediate language is designed.

In order to solve the conciseness and readability problem mentioned earlier, the intermediate language could be designed to function as a concrete ASCII syntax for RT-Tester models. With this method, rather than having two separate translation steps, the second step would instead be a simple parsing step, where the RT-Tester model is parsed from a concrete ASCII syntax to the corresponding C++ objects.
This usefulness of such a concrete ASCII syntax can be easily demonstrated by the following comparison. Consider the following intermediate language expression:

```
(x == 1 && x' == 2) || (y == 2 && y' == 3)
```

This expression is much more concise and readable compared to the corresponding expression tree construction in RT-Tester:

```
RttTgenExpTree expr =
  new RttTgenExpTree("||",INFIXOPERATOR,BOOLOR);
RttTgenExpTree exprLeft =
  new RttTgenExpTree("&&",INFIXOPERATOR,BOOLAND);
expr.setLeft(exprLeft);
RttTgenExpTree exprRight =
  new RttTgenExpTree("&&",INFIXOPERATOR,BOOLAND);
expr.setRight(exprRight)
```

```
RttTgenExpTree exprLeftLeft =
  new RttTgenExpTree("==",INFIXOPERATOR,BOOLAND);
RttTgenExpTree exprLeftRight =
  new RttTgenExpTree("==",INFIXOPERATOR,BOOLAND);
exprLeft.setLeft(exprLeftLeft);
exprLeft.setRight(exprLeftRight);
RttTgenExpTree x
  = new RttTgenExpTree("x",IDENTIFIER,NAMEX);
x.setVersion(0);
exprLeftLeft.setLeft(x);
exprLeftLeft.setRight(new RttTgenExpTree(1ll));
RttTgenExpTree xPrime =
  new RttTgenExpTree("x'",IDENTIFIER,NAMEX);
x.setVersion(1);
exprLeftRight.setLeft(x);
exprLeftRight.setRight(new RttTgenExpTree(2ll));

RttTgenExpTree exprRightLeft =
  new RttTgenExpTree("==",INFIXOPERATOR,BOOLAND);
RttTgenExpTree exprRightRight =
  new RttTgenExpTree("==",INFIXOPERATOR,BOOLAND);
exprRight.setLeft(exprRightLeft);
exprRight.setRight(exprRightRight);
RttTgenExpTree y =
  new RttTgenExpTree("y",IDENTIFIER,NAMEX);
y.setVersion(0);
exprRightLeft.setLeft(y);
exprRightLeft.setRight(new RttTgenExpTree(2ll));
RttTgenExpTree yPrime =
  new RttTgenExpTree("y'",IDENTIFIER,NAMEX);
x.setVersion(1);
exprRightRight.setLeft(y);
exprRightRight.setRight(new RttTgenExpTree(3ll));
```

In this particular case, the intermediate language expression is very similar to its corresponding RSL representation. One could therefore argue, that it is not clear what is to be gained by the intermediate step. However, as it will be discussed in Section 4.3, there are several important structural differences between RSL and RT-Tester, where the concrete ASCII syntax can help visualise the resulting RT-Tester model.

There is an additional advantage if the intermediate language is designed as described above, namely that the parsing step when going from the intermediate language to RT-Tester should be very straightforward to implement, since the structure of the intermediate language will be based on RT-Tester. The implementation is also aided by the fact, that there already exists a parser within RT-Tester, which can parse LTL expressions written in an ASCII syntax to the corresponding objects in RT-Tester. This parser can then be reused in the implementation of the complete intermediate language parser.

3.2.2.1 Aiding future RSL translations

Translating RSL specifications to an intermediate language also offers an interesting opportunity.
The intermediate language could be designed in such a way, that the entire RSL translator could be reused in other projects similar to this one. There exists many different model checking tools, and one could easily imagine future projects which deal with the translation from RSL to such another tools.
If the intermediate language in this project is designed to act as a universal language for similar models, the only thing left to develop in future projects would be the parsing from the intermediate language to the model checker.
This idea is illustrated below in Figure 3.3:

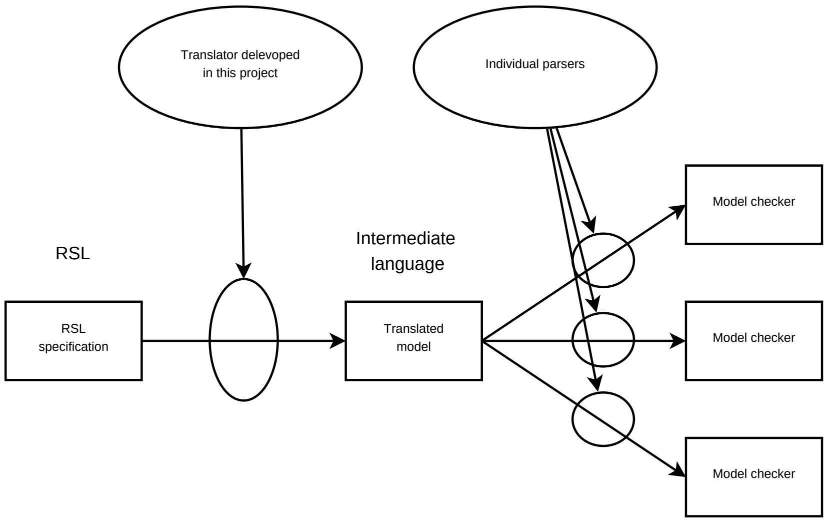

Figure 3.3: Reusing the intermediate language

3.3 Translation tools

There exists a wide array of tools that can aid with language translation. Here is a short description of some of the more appealing tools that has been considered for this project:

3.3.1 rsltc and Gentle

The RSL Type Checker (rsltc) is a set of useful tools specifically for RSL [2]. It has many useful features, such as translating RSL to various other languages, but the two most relevant tools in relation to this project, is the type checker and the construction of abstract syntax trees. The type checker can be used to check the static correctness of RSL specifications before they are translated, and the existing construction of abstract syntax trees can be reused to save the time it would take to implement this from scratch.

The rsltc tool set is made using Gentle, which is a compiler construction system [4]. If one were to make an extension to rsltc, it would be obvious to also use Gentle for this, so the rsltc source code can be amended directly. Like most other tools, Gentle allows the use of high-level descriptions when generating compilers or translators, which makes the tool relatively easy to use.

3.3.2 ANTLR

ANTLR (Another Tool for Language Recognition) is one of the most popular tools for constructing recognizers, interpreters, compilers and translators [12]. The tool works by taking a context-free grammar specifying a language as input, and generates the code for a recognizer in a selection of programming languages. A recognizer simply checks whether an input follows a certain grammar, so in order to do something more useful with the language, actions can be attached to elements of the grammar. ANTLR also provides a consistent notation for specifying lexers and parsers, which can also be generated by the tool.

ANTLR is one of the more easy language recognition tools to use, and is widely used both in industry and academia. But besides its usability, there are no distinguishing features in ANTLR relevant for this project compared to other tools.

3.3.3 Lex and Yacc

Lex and Yacc [15] are a set of programs often used in conjunction with each other. Lex is used to generate lexers and Yacc is used to generate parsers.
There exists a variety of different implementations of Lex and Yacc, also for a variety of programming languages. It is worth noting, that the existing parsers in RT-Tester already use Lex and Yacc.

3.4 The chosen method

The chosen method for this project, is to translate using a concrete ASCII syntax of RT-Tester models as an intermediate language, and parsing this language into an RT-Tester representation. The focus of the intermediate language is first and foremost to act as a concise and readable representation, which can easily be parsed into the corresponding RT-Tester model. However, it is also with the idea that the intermediate language can be reused in other future projects, where RSL specifications are translated and used by other model checking systems than RT-Tester.

The translation into the intermediate language will be implemented by using Gentle to extend the rsltc tool set, and thereby reusing key parts of the tool, namely the type checker and the construction of abstract syntax trees. For the intermediate language parser, the existing LTL parser will be extended using Lex and Yacc to cover the parsing of entire models, rather than just LTL formulae. This method of approach is illustrated below in Figure 3.4.

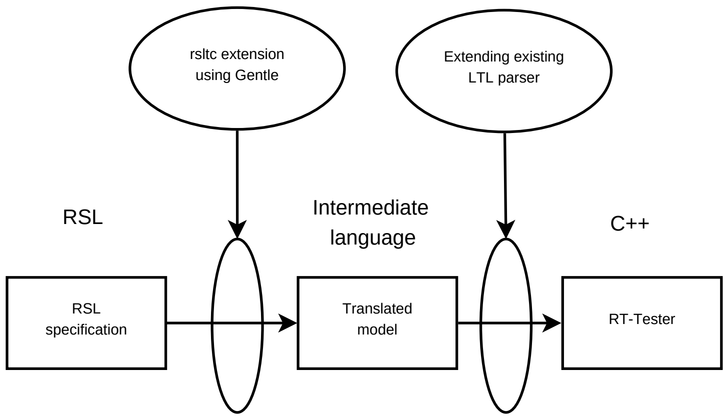

Figure 3.4: Chosen translation method

By using the intermediate language, it should allow for easier testing and debugging, since any RSL specification can be manually replicated in the intermediate language and compared with the implemented translation. Also, by simply extending rsltc, rather than developing a translator from scratch with another tool, it should be possible to quickly develop a translator for a small subset of RSL and then extending it incrementally, thereby easing the development process.

Chapter 4

Design choices and analysis

In this chapter, the overall design of the translation from RSL to RT-Tester will be presented and discussed. The following topics form the main content of this chapter:

- A presentation of the RSL constructs that can be translated in this project, and a discussion of why certain constructions have been left out.
- A description of the structure of the intermediate language, and how it relates to the structure of RT-Tester models.
- An analysis of the RSL translation, and a presentation of the system of inference rules, that defines the translation from RSL to the intermediate language.
- A short description of how the intermediate language is parsed into a model in RT-Tester.
- A discussion of variable bounds and how they can be handled in a number of different ways.
- An outline of how the number of transition steps used in the model checker can be provided by the user.

4.1 Translatable subset of RSL

In this project, only a small subset of the constructions available in RSL is accepted by the translator. This section will outline this subset, and justify why certain constructions have been left out. What the different constructions are translated to, is described in Section 4.3.

Please note that many definitions in this section has been taken from [1].

4.1.1 Declarations

The following declarations are accepted by the translator:

4.1.1.1 Scheme declarations

Scheme declarations are accepted by the translator. Formal scheme parameters are not accepted.
Accepted scheme declarations have the form:

scheme optionalName = class_expression

4.1.1.2 Type declarations

Type declarations are accepted by the translator.
Accepted type declarations have the form:

type
type_definition_list

Type declarations may only contain *Abbreviation definitions* and *Variant definitions.*

Abbreviation type definitions
Accepted abbreviation type definitions have the form:

id = type_expression

Variant type definitions
Accepted variant type definitions have the form:

id == variant_1 | ... | variant_n

Only constant variants are accepted.

4.1.1.3 Value declarations

Value declarations are accepted by the translator.
Accepted value declarations have the form:

value
  value_definition_list

Value declarations may only contain *Explicit value definitions* and *Explicit function definitions.*

Explicit value definitions
Accepted explicit value definitions have the form:

binding : type_expression = value_expression

Explicit function definitions
Accepted explicit function definitions have the form:

id : type_expression_1 $\times$ ... $\times$ type_expression_n $\rightarrow$
type_expression
id(binding_1,...,binding_n) $\equiv$ value_expression

Recursive and partial function definitions are not accepted.

4.1.1.4 Transition system declarations

Transition system declarations are accepted by the translator.
Accepted transition system declarations have the form:

```
transition_system [ name ]
  local
    variable_definition_list
  in
    transition−rule_definition_1
    ▯
    ...
    ▯
    transition−rule_definition_n
```

where $n \geq 1$ **Variable definitions**
Accepted variable definitions have the form:

```
id : type_expression := value_expression
```

Transition rule definitions
Accepted transition rule definitions have the form:

```
[ name ] value_expression ⟶ value_expression
```

4.1.1.5 LTL assertion declarations

LTL assertion declarations are accepted by the translator.
Accepted LTL assertion declarations have the form:

```
ltl_assertion
  assertion_definition_list
```

LTL assertion definitions
Accepted LTL assertion definitions have the form:

```
[ name ] id ⊢ ltl_formula
```

4.1.2 Class Expressions

The following class expressions are accepted by the translator:

4.1.2.1 Basic class expressions

Basic class expressions are accepted by the translator.
Accepted basic class expressions have the form:

```
class
  declaration_list
end
```

4.1.3 Type Expressions

The following type expressions are accepted by the translator:

4.1.3.1 Type names

Type names are accepted by the translator.
Accepted type names have the form:

```
name
```

where *name* must represent a type.

4.1.3.2 Type literals

Type literals are accepted by the translator.
The type literals accepted by the translator are:

- Int
- Real

- Bool

4.1.3.3 Subtype expressions

Subtype expressions are accepted by the translator.
Accepted subtype expressions have the form:

name = {|id : type_literal • value_expression |}

4.1.4 Value Expressions

The following value expressions are accepted by the translator:

4.1.4.1 Variable names

Variable names are accepted by the translator.
Accepted variable names have the form:

name

where *name* must represent a variable.

4.1.4.2 Value names

Value names are accepted by the translator.
Accepted value names have the form:

name

where *name* must represent a value.

4.1.4.3 Value literals

Value literals are accepted by the translator.
The value literals accepted by the translator are:

- Int literals
- Real literals
- Bool literals

4.1.4.4 Function application expressions

Function application expressions are accepted by the translator.
Accepted function application expressions have the form:

id(value_expression_list)

4.1.4.5 Value infix expressions

Value infix expressions are accepted by the translator.
Accepted value infix expressions have the form:

value_expression_1 infix_operator value_expression_2

where *infix_ operator* is one of the following operators:

=	equality
+	addition
−	subtraction
∗	multiplication
/	division
<	less than
>	greater than

$\leq$: less than or equal to

$\geq$: greater than or equal to

$\vee$: boolean disjunction

$\wedge$: boolean conjunction

4.1.4.6 Value prefix expressions

Value prefix expressions are accepted by the translator, but only for the negation operator.
Accepted value prefix expressions have the form:

```
∼value_expression
```

4.1.4.7 If expressions

If expressions are accepted by the translator.
Accepted if expressions have the form:

```
if logical−value_expression
  then value_expression_1
    optional−elsif_branch_list
  else value_expression_2
end
```

where *elsif_branch* has the form:

```
elsif logical−value_expression then value_expression
```

If expressions are only allowed as part of *transition rule definitions* and as part of *explicit function definitions* where the function returns a boolean value. This is due to the way variables are represented in RT-Tester models (more on this in Section 4.3.3).

4.1.4.8 Case expressions

Case expressions are accepted by the translator.
Accepted case expressions have the form:

case value_expression **of** case_branch_list **end**

where *case_ branch* has the form:

pattern → value_expression

At least one of the case branches must use the wildcard pattern '_'. Like if expressions, case expressions are only allowed as part of *transition rule definitions* and as part of *explicit function definitions* where the function returns a boolean value. This is due to the way variables are represented in RT-Tester models (more on this in Section 4.3.3).

4.1.5 Discarded constructions

In general, the reason why this project only deals with the constructions listed earlier, is simply due to time constraints. RSL is far too comprehensive to be translated in its entirety within the scope of this project.
However, due to the nature of RT-Tester, there are some RSL constructions that would be particularly difficult to translate, and some that are outright impossible.

4.1.5.1 Difficult constructions

Lists
The way lists function in RSL is very similar to how arrays are represented in RT-Tester models. It would therefore be natural to use the existing structure for arrays, when translating RSL lists into RT-Tester. However, the SMT solver which perform the model checking in RT-Tester is currently not equipped to handle arrays. So even if the translation of lists was implemented using arrays, there is no way to make use of the constructs in the solver.
If the SMT solver should be updated to handle arrays in the future, it would be obvious to include translation of lists in any further development of this project.

Sets and maps
Sets and maps are common in RSL specifications, and are very useful constructions when modelling systems. It would therefore be preferred to include them in this project. However, this would be a very complicated task.
First of all, there are no similar constructions in RT-Tester, so any direct translation is not an option. If only sets and maps of constant size were considered, it would be possible to create simple typed variables in RT-Tester for each element in the RSL construction. As long as a fixed number of variables can be created before verifying the model, this method would be an option. Constant sized sets and maps can still be useful for modelling, and this could therefore a logical feature in any further development of this project.
The major problem comes, when the size of the sets and maps are dynamic, i.e. elements are added and removed when transition rules are taken. Here it is no longer an option to just create a number of simple typed variables, and it would require a lot of changes to the implementation of the SMT solver in RT-Tester, in order to be able to create and delete variables in the symbol table during model checking. The implementation of the solver is fairly complex, and it has therefore been deemed too time-consuming to attempt to make it compatible with dynamic data structures.

4.1.5.2 Impossible constructions

Axioms
Axiomatic declarations and definitions are a way of properties in RSL, which must hold in the model. There are no corresponding mechanism in RT-Tester for defining axioms, which makes sense seeing as the whole purpose of model checking in the first place, is to prove whether the model satisfies some given properties. Therefore, the idea is that rather than defining the properties as axioms (invariants), the properties should be written as LTL formulae, which are then model checked in RT-Tester.

Channels
Channel declarations and definitions are used in RSL to introduce concurrency, and there is currently no way to represent concurrency when model checking using RT-Tester.

Chaos
The expression **Chaos** is used in RSL to represent chaotic behaviour of programs, such as a program that never terminates. Though the behaviour of Chaos is well defined in RSL (for instance when used in if expression or when used in boolean connectives), there is no clear way of representing this in RT-Tester.

Even if there was a corresponding mechanism is RT-Tester, Chaos is very rarely used in RSL specification, and how not be a priority to include in the translator.

4.1.6 Example of a translatable RSL specification

Below is an example of an RSL specification which uses most of the constructions that are accepted by the translator in this project. To get a gradual introduction to the translatable constructions, see Appendix A.1, where six RSL specification examples are given along with their translation, ending with the example given below.

This example specifies a simple airport system.
There are two types defined in the specification. *Weather*, which is a type variant listing the possible weather conditions, and *nat* which is a subtype defining all natural numbers. The basic type Nat is not one of the accepted type literals, so nat is manually defined instead.

There are two variables in the specification, namely *numberOfPlanes* of type *nat* which tracks the number of planes in the airport, and *weatherConditions* of type *Weather* which tracks the current weather. There is also a constant value *planeCapacity* of type *nat* which restricts the number of planes allowed in the airport.

There are two function definitions in the specification.
The function *hasFreeCapacity* checks whether the airport can handle an arriving flight, and it takes the number of planes, the plane capacity and the weather conditions as parameters.
The function *hasAvailablePlanes* checks whether the airport can handle a flight departure, and it takes the number of planes and weather conditions as parameters.

There are six transition rules in the specification.
The first transition rule *planeArrival* uses the function *hasFreeCapacity* as guard and increments the number of planes if possible.
The second transition rule *planeDeparture* uses the function *hasAvailablePlanes* as guard and decrements the number of planes if possible.
The final four transitions simply change the weather conditions to a value different than the current one.

```
scheme Airport6 =
class
   type
    Weather == Sunny | Cloudy | Stormy | Hurricane,
    nat = {| n : Int • n ≥ 0 |}
   value
    planeCapacity : nat = 150,
    hasFreeCapacity : nat × nat × Weather → Bool
    hasFreeCapacity(p,c,w) ≡
      if w = Stormy ∨ w = Hurricane then false else p < c end,
    hasAvailablePlane : nat × Weather → Bool
    hasAvailablePlane(p,w) ≡
      case w of
       Stormy → false,
       Hurricane → false,
       _ → p > 0
      end
   transition_system [TS]
    local
      numberOfPlanes : nat := 100,
      weatherConditions : Weather := Sunny
    in
      [planeArrival]
      hasFreeCapacity(numberOfPlanes,planeCapacity,weatherConditions) ⟶
         numberOfPlanes′ = numberOfPlanes + 1
      ⌷
      [planeDeparture]
      hasAvailablePlane(numberOfPlanes,weatherConditions) ⟶
         numberOfPlanes′ = numberOfPlanes − 1
      ⌷
      [SunnyWeather] weatherConditions ≠ Sunny ⟶
         weatherConditions′ = Sunny
      ⌷
      [CloudyWeather] weatherConditions ≠ Cloudy ⟶
         weatherConditions′ = Cloudy
      ⌷
      [StormyWeather] weatherConditions ≠ Stormy ⟶
         weatherConditions′ = Stormy
      ⌷
      [HurricaneWeather] weatherConditions ≠ Hurricane ⟶
         weatherConditions′ = Hurricane
    end
```

```
  ltl_assertion
   [CapacityConstraint] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
      numberOfPlanes ≥ 0)
end
```

4.2 Design of the intermediate language

The purpose of the intermediate language, as established in the previous chapter, is to act like a bridge between RSL and RT-Tester, such that the intermediate language displays the same information as contained in an RT-Tester model, but in an easily readable language. As such, the intermediate language should be designed in such a way, that the process of parsing the intermediate language into a model in RT-Tester is as straightforward as possible. To achieve this, the structure of the intermediate language should be a reflection of the structure of models in RT-Tester.

4.2.1 Structure

RT-Tester is written in the C++ language. As such, an RT-tester model is really a collection of objects defined in C++. At the highest level of abstraction, the SMT solver used in RT-Tester must be provided four different objects, in order to perform model checking:

- **A symbol table**, where all the variable types, variable instances and functions are defined.
- **Initial values of variables**, given as an expression in propositional logic.
- **A transition relation**, given as an expression in propositional logic.
- **LTL formulae**, representing the properties to be verified.

In order to mirror the structure of RT-Tester models and make parsing easier, the intermediate language structure is divided into four parts in a similar fashion.

The first part contains the information needed to create the symbol table in RT-Tester, i.e. all custom type, variable and function definitions. The two keywords 'SYM_TABLE_DECL' and 'SYM_TABLE_DECL_END' are used to delimit this part:

```
SYM_TABLE_DECL
```
symboltable definitions
```
SYM_TABLE_DECL_END
```

The second part contains the initial values of all variables. The two keywords 'INIT_VAL' and 'INIT_VAL_END' are used to delimit this part:

```
INIT_VAL
```
initial values
```
INIT_VAL_END
```

The third part contains the transition relation, and the two keywords 'TRANS_REL' and 'TRANS_REL_END' are used as delimiters:

```
TRANS_REL
```
transition relation
```
TRANS_REL_END
```

The final part contains the property specifications to be verified, and is delimited by the keywords 'PROP_SPEC' and 'PROP_SPEC_END':

```
PROP_SPEC
```
property specifications
```
PROP_SPEC_END
```

Of course, this is just the superficial structure of the intermediate language. The syntax of the language will be defined in the following section, and the translation from RSL will be discussed and presented as a system of inference rules in Section 4.3.

4.2.2 BNF grammar

The exact syntax of the intermediate language, is defined by the following BNF grammar. The analysis that lead to this syntax and its relation to RSL is described in Section 4.3.
Note that the regular expression operator '*' is used to represent any number of repetitions, and '\n' is used to represent a line break.

```
<grammar> ::= "SYM_TABLE_DECL \n" <sym_tab_defs>
  "SYM_TABLE_DECL_END \n" "INIT_VAL \n"
  <init_vals> "INIT_VAL_END \n" "TRANS_REL \n" <trans_rel>
  "TRANS_REL_END \n n" "\n PROP_SPEC \n" <prop_specs>
  "PROP_SPEC_END"

<sym_tab_defs> ::= <sym_tab_def> "\n" (<sym_tab_def> "\n")*

<sym_tab_def> ::= <var_def> | <const_def> | <fun_def> |
  <type_def>

<var_def> ::= <type> " " <id>

<const_def> ::= "const " <id> " " <id> " == " <val_expr>

<fun_def> ::= <id> " " <id> " (" (<id> " " <id>)* ")
  return" <val_expr> ""

<type_def> ::= <variant_type_def> | <abbrev_type_def> |
  <sub_type_def>

<variant_type_def> ::= <id> " == " <id> (" | " <id>)*

<abbrev_type_def> ::= <id> " == " <id>

<sub_type_def> ::= <id> " == " <id> " " <id> " where "
  <val_expr>

<init_vals> ::= <init_val> "\n" (<init_val> "\n")*

<init_val> ::= <id> " == " <val_expr>

<trans_rel> ::= <bool_expr> ("|| \n" <bool_expr>)*

<prop_specs> ::= <prop_spec> "\n" (<prop_spec> "\n")*

<prop_spec> ::= bool_expr | "Globally[" <prop_spec> "]" |
  "Finally[" <prop_spec> "]" | "Next[" <prop_spec> "]" |
  "[" <prop_spec> "] Until [" <prop_spec> "]"

<val_expr> ::=  <literal> | <val_expr> <arith_op> <val_expr> |
                <id> "(" <val_expr>+ ")"

<bool_expr> ::= <bool_literal> | <val_expr> <bool_op> <val_expr> |
                <prefix_op> <bool_expr>
```

```
<literal> ::= <id> | <digit>* | <bool_literal>

<bool_literal> ::= "true" | "false"

<infix_op> ::= <arith_op> | <bool_op>

<prefix_op> ::= "!"

<arith_op> ::= "+" | "-" | "*" | "/"

<bool_op> ::= "==" | "<=" | ">=" | "<" | ">" | "&&" | "||"

<id> ::= <letter> | <char>*

<char> ::= <letter> | <digit> | "_" | "'"

<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" |
             "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" |
             "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" |
             "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" |
             "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
             "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
             "W" | "X" | "Y" | "Z"

<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" |
            "8" | "9"
```

4.2.3 Example of intermediate language model

The following intermediate language model is the translation of the RSL example from Section 4.1.6:

```
SYM_TABLE_DECL
Weather == Sunny | Cloudy | Stormy | Hurricane
nat == int n where n >= 0
const nat planeCapacity == 150
bool hasFreeCapacity (nat p,nat c,Weather w)
  {return ((w == Stormy || w == Hurricane && false) ||
           (!(w == Stormy || w == Hurricane) && p < c))}
bool hasAvailablePlane (nat p,Weather w)
```

```
  {return (w == Stormy && false) || (w == Hurricane && false) ||
          (!(w == Hurricane) && !(w == Stormy) && p > 0)}
nat numberOfPlanes
Weather weatherConditions
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
weatherConditions == Sunny
INIT_VAL_END

TRANS_REL
(hasFreeCapacity(numberOfPlanes,planeCapacity,weatherConditions)
  && numberOfPlanes' == numberOfPlanes + 1
  && weatherConditions' == weatherConditions) ||
(hasAvailablePlane(numberOfPlanes,weatherConditions)
  && numberOfPlanes' == numberOfPlanes - 1
  && weatherConditions' == weatherConditions) ||
(weatherConditions != Sunny && weatherConditions' == Sunny &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Cloudy && weatherConditions' == Cloudy &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Stormy && weatherConditions' == Stormy &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Hurricane &&
  weatherConditions' == Hurricane &&
  numberOfPlanes' == numberOfPlanes)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

4.3 Translation analysis

In this section, the translation from the translatable RSL subset defined in Section 4.1 to the intermediate language will be analysed, and a system of inference rules defining the translation will be presented. The exact syntax of all intermediate language constructions can then be inferred based on this system. Examples of this translation can be seen in Appendix A.1.

4.3.1 The general approach

The general approach to this translation, is to make intermediate language as similar to an RT-Tester model as possible. This means that whenever there is a conceptual difference between RSL models and RT-Tester models, this should be resolved in the first translation step to the intermediate language, rather than in the subsequent parsing step. In doing this, the intermediate language becomes a readable representation of an RT-Tester model. This is very useful since an RT-Tester model in and of itself is a collection of C++ objects, with very little transparency in relation to the model it represents.

In keeping with this approach, the syntax of some of the RSL constructions must be changed in the translation, so that the intermediate language matches the syntax used in RT-Tester. This way, the existing LTL parser in RT-Tester can also be reused as a parser for the intermediate language without any changes being made to it.
Most of these translations are trivial, such as the boolean operators $\wedge$ and $\vee$ being translated to && and $||$. There are a few non-trivial translations, which will be analysed in the Sections 4.3.2, 4.3.3 and 4.3.4.

Another benefit to this approach, is that the translator from RSL to the intermediate language may be reused in other similar project, where RSL specifications are being model checked by some model checking system other than RT-Tester. Of course, this is assuming that such systems have a similar structure to that of RT-Tester.

An important thing to note, is that the intermediate language is not designed to act as an independent or consistent language. There will be made no effort in the following system of inference rules to do any kind of type checking of the constructions being translated. Instead, it is the idea that the existing RSL type checker (rsltc) will be used to check any model before translation is attempted. Additionally, the rsltc can also be used to generate so-called confidence conditions. This does not find clear cut errors similar to the type checker, but it will point to potential issues in the model (such as a function which divides by zero for certain parameters). These confidence conditions can then be used by the user, to manually check the correctness of the model, before attempting to translate it.

4.3.2 Transition rules

There are two important differences between how a transition rule is represented in RSL and in RT-Tester.

The first difference is in the structure of the transition rule. In RSL a transition rule has the following form:

$$guard \rightarrow update$$

Here the guard is a boolean expression, which enables the transition rule to be available, and the update is an expression updating the value of some variables. A concrete example of this in RSL syntax could be:

$$x = 1 \wedge y = 1 \longrightarrow x' = 2 \wedge y' = 2$$

In RT-Tester, the transition rules have following form:

$$guard \;\&\&\; update$$

The previous example then translates to the following in RT-Tester:

$$x = 1 \;\&\&\; y = 1 \;\&\&\; x' = 2 \;\&\&\; y' = 2$$

When multiple transition rules are combined to form the transition relation in RT-Tester, all the transition rule expressions are put into a single disjunction of transition rules, as shown below:

$$\begin{array}{l} \;\; guard_1 \wedge update_1 \\ [] \\ \;\; guard_2 \wedge update_2 \\ [] \\ \;\; \ldots \\ [] \\ \;\; guard_n \wedge update_n \end{array}$$

These rules translate to following transition relation in RT-Tester:

$$(guard_1 \;\&\&\; update_1) \;||\; (guard_2 \;\&\&\; update_2) \;||\; \ldots \;||\; (guard_n \;\&\&\; update_n)$$

The second difference between transition rules in RSL and RT-Tester is more conceptual. In RSL, it is only necessary to list the variables which are being

updated, when the transition rule is used. It can then be inferred, that the value of all other variables in the model remain unchanged. However, this is not the case in RT-Tester. Here it must be explicitly stated what the value of each variable is after using a given transition rule.
An example of this can be seen in the following RSL transition system:

```
transition_system [name]
  local
    x : Int := 1,
    y : Int := 2
  in
    x = 1 ⟶ x′ = 2
    ▯
    y = 2 ⟶ y′ = 3
```

The two transition rules in this example would translate to the following in RT-Tester:

$$(x == 1 \ \&\& \ x' == 2 \ \&\& \ y' == y) \ || \ (y == 2 \ \&\& \ y' == 3 \ \&\& \ x' == x)$$

4.3.3 If expressions

If expressions do not have a directly corresponding construction in the RT-Tester model language. There is such a construction within the SMT solver used in RT-Tester, but this is not used, since the RT-Tester model should work regardless of which SMT solver is used for model checking.
So instead, the way if expression then are translated, is by considering what the underlying logical statement of such an expression is. Take for instance the following if expression, which could be used as the guard in a transition rule:

if x = 0 **then** y = 1 **else** y = 2 **end**

This will then be rewritten to an equivalent logical statement:

$(\mathrm{x} = 0 \wedge \mathrm{y} = 1) \vee (\sim(\mathrm{x} = 0) \wedge \mathrm{y} = 2)$

However, rewriting if expression in such a manner may produce problems when if expressions are used as part of a larger value expression. Consider the following value expression:

$$x' = (\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 2 \textbf{ end})$$

This is a perfectly valid expression in RSL, but when the if expression is rewritten to a separate logical statement, is becomes the following:

$$(x = 0 \wedge 1) \vee (x = 1 \wedge 2)$$

Since 1 and 2 are not boolean expressions, this is no longer a useful expression in RT-Tester.
The solution to this, is to consider the entire value expression that surrounds the if expression. We can then first rewrite the value expression in such a way, that the if expression is the outermost construction, before it is translated into a logical statement.
Consider the value expression from earlier:

$$x' = (\textbf{if } x = 0 \textbf{ then } 1 \textbf{ else } 2 \textbf{ end})$$

This can be rewritten into the following equivalent expression:

$$\textbf{if } x = 0 \textbf{ then } x' = 1 \textbf{ else } x' = 2 \textbf{ end}$$

By having the if expression as the outermost construction, the expression can now be translated to an equivalent logical statement.
However, if the value expression that includes the if expression does not evaluate to a boolean value, then it cannot be rewritten to a meaningful logical statement, which is why the translator only accepts if expressions when used as part of transition rule definitions and explicit function definitions with boolean return values.

The inference rules that define this rewriting process can be found in Section 4.3.5.26, and the inference rules that define the translation of if expressions can be found in Section 4.3.5.24.
A full example using these rules can be found in Appendix A.2.

4.3.4 Case expressions

Like if expressions, case expressions does not have a corresponding construction in RT-Tester.

However, both expression types are really just two ways of representing the same expression. Consider the following if and case expressions, and note that they are equivalent:

```
if x = 0 then 1
    elsif x = 1 then 2
else 3 end
```

```
case x of
    0 → 1,
    1 → 2,
    _ → 3
end
```

So instead of having inference rules which translate both if and case expression to the intermediate language, all case expressions will be rewritten into an equivalent if expression, and then translated using the translation rules for if expressions.

Because case expressions are rewritten to if expression, case expression also share the same limitation of only being accepted in transition rule definitions and function definitions with boolean return values.

4.3.5 System of inference rules for RSL translation

In the following system of inference rules, the translation from RSL to the intermediate language is defined by the operator $\rhd$.

Contexts
When translating an RSL specification to the intermediate language, it is often the case the translation result from a declaration is used in multiple parts of the intermediate language.
An example of this is the transition system declaration. This declaration contains information relevant for both the symbol table, initial value and transition relation part of the intermediate language.
Because of this, the concept of *contexts* has been introduced in the following system of inference rules. Then by using four different context keyword and applying them to the appropriate rules, it is possible to produce the desired

output based on which part of the intermediate language is being created. The following context keywords are used:

st : symbol table

iv : initial values

tr : transition relation

ps : property specifications

Contexts are placed to the left of the symbol $\vdash$. So $c \vdash e \rhd e'$ means that e translates to *e'* in context c. Some RSL constructions only contribute to the translation in a given context. An example of this could be a type declaration, which only contributes in the context 'st' (when defining the symbol table). This is reflected by adding a rule such as the following, where ϵ represents the empty translation result:

$$\frac{c \neq st}{c \vdash type\_decl \rhd \epsilon}$$

Because many of the RSL constructs should be translated the same regardless of the context (e.g. operators), the context notation is only used for rules where is makes a difference for the translation. Such rules are simply denoted without using context, as such:

$$\frac{}{e \rhd e'}$$

This means that e translates to *e'* regardless of the given context.

Besides the contexts mentioned earlier, the context 'vs' is also used, but for another purpose. When translating transition rules, the variables which are not updated by the rule must be part of the translation. Therefore, the context 'vs' is used to collect the identifiers of all variable definitions. The context can then be passed on to the inference rule for transition rules, where it is compared with the set of variables that are updated in the transition rule.

Term variables

In order to make the rules more concise, some term variables will be used throughout the system of inference rules. These term variables only matches constructs in to the subset of RSL defined in Section 4.1.

s : Any string, typically an identifier.

e : A value literal or an identifier, i.e. the smallest component of any value expression.

op : An operator.

ve : A value expression, meaning any combination of literals, identifiers, operators, if expressions or case expressions. The term e is also a value expression.

be : A boolean expression, i.e. a value expression that evaluates to a boolean value.

type : A type name.

List notation
Also note, that a shorthand notation '..' is used whenever a construction contains an unknown number of terms. For instance, a variant definition may contain any number enumerators, and is therefore represented as the following:

$$type == e_1 \mid .. \mid e_n$$

When using the translation operator $\triangleright$ with this notation, it corresponds to translating each of the elements separately. This way, the following two constructions are equivalent when $n = 4$:

$$\frac{e_1 \triangleright e'_1,\ e_2 \triangleright e'_2,\ e_3 \triangleright e'_3,\ e_4 \triangleright e'_4}{e_1 \mid e_2 \mid e_3 \mid e_4 \ \triangleright\ e'_1 \mid e'_2 \mid e'_3 \mid e'_4}$$

$$\frac{e_1,\ ..\ ,\ e_n \triangleright e'_1,\ ..\ ,\ e'_n}{e_1 \mid .. \mid e_n \ \triangleright\ e'_1 \mid .. \mid e'_n}$$

Normal form
As described in Sections 4.3.3 and 4.3.4, value expressions containing if or case expression are rewritten such that case expressions become if expressions, and if expressions become the outermost construction.
Value expression that adhere to these requirements is said to be in *normal form*. A value expression in normal form must adhere to the following rules:

- The value expression does not contain case expressions.
- The value expression does not contain if expression with elsif branches.
- The value expression does not contain if expression as part of an infix, prefix or function application expression.

This concept of normal form is used in some of the following inference rules, to determine whether further rewriting of a value expression is necessary.

Here are some examples of value expression *not* in normal form:

```
x = 1 + if true then 2 else 3 end

x = if y = 1 then 1 else 2 end + if y = 2 then 3 else 4 end
```

Here are the same expression rewritten to normal form:

```
if true then x = 1 + 2 else x = 1 + 3 end

if y = 1 ∧ y = 2 then x = 1 + 3 else
   if y = 1 ∧ ∼(y = 2) then x = 1 + 4 else
      if ∼(x = 1) ∧ y = 2 then x = 2 + 3 else x = 2 + 4
      end
   end
end
```

An example of rewriting value expressions using the system of inference rules can be found in Appendix A.2.

4.3.5.1 Scheme declarations

The following inference rules define the translation operator $\triangleright$ for scheme declarations:

$$\frac{\begin{array}{l} st \vdash scheme \triangleright sym\_tab\_defs,\ vs \\ iv \vdash scheme \triangleright init\_vals \\ tr, vs \vdash scheme \triangleright trans\_rel \\ ps \vdash scheme \triangleright prop\_specs \end{array}}{\begin{array}{l} scheme \triangleright \\ sym\_tab\_defs \\ init\_vals \\ trans\_rel \\ prop\_specs \end{array}} \tag{4.1}$$

where *scheme* is a scheme declaration

$$\frac{st \vdash decl_1,\ ..\ ,\ decl_n \rhd decl'_1,\ vs_1,\ ..\ ,\ decl'_n,\ vs_n}{\begin{array}{l} st \vdash \textbf{scheme}\ s = \\ \qquad \textbf{class} \\ \qquad\quad decl_1 \\ \qquad\quad .. \\ \qquad\quad decl_n \\ \qquad \textbf{end} \\ \rhd \\ SYM\_TABLE\_DECL \\ decl'_1\ ..\ decl'_n \\ SYM\_TABLE\_DECL\_END,\ vs_1 \cup .. \cup vs_n \end{array}} \tag{4.2}$$

where $decl_i$ is a type declaration, value declaration, transition system declaration or LTL assertion declaration

$$\frac{iv \vdash decl_1,\ ..\ ,\ decl_n \rhd decl'_1,\ ..\ ,\ decl'_n}{\begin{array}{l} iv \vdash \textbf{scheme}\ s = \\ \qquad \textbf{class} \\ \qquad\quad decl_1 \\ \qquad\quad .. \\ \qquad\quad decl_n \\ \qquad \textbf{end} \\ \rhd \\ INIT\_VALS \\ decl'_1\ ..\ decl'_n \\ INIT\_VALS\_END \end{array}} \tag{4.3}$$

where $decl_i$ is a type declaration, value declaration, transition system declaration or LTL assertion declaration

$$\frac{tr,\ vs \vdash decl_1,\ ..,\ decl_n \rhd decl'_1,\ ..,\ decl'_n}{\begin{array}{l} tr,\ vs \vdash \textbf{scheme}\ s = \\ \qquad\quad \textbf{class} \\ \qquad\qquad decl_1 \\ \qquad\qquad .. \\ \qquad\qquad decl_n \\ \qquad\quad \textbf{end} \\ \rhd \\ TRANS\_REL \\ decl'_1\ ..\ decl'_n \\ TRANS\_REL\_END \end{array}} \tag{4.4}$$

where $decl_i$ is a type declaration, value declaration, transition system declaration or LTL assertion declaration

$$\frac{ps \vdash decl_1,\ ..,\ decl_n \rhd decl'_1,\ ..,\ decl'_n}{\begin{array}{l} ps \vdash \textbf{scheme}\ s = \\ \qquad\quad \textbf{class} \\ \qquad\qquad decl_1 \\ \qquad\qquad .. \\ \qquad\qquad decl_n \\ \qquad\quad \textbf{end} \\ \rhd \\ PROP\_SPEC \\ decl'_1\ ..\ decl'_n \\ PROP\_SPEC\_END \end{array}} \tag{4.5}$$

where $decl_i$ is a type declaration, value declaration, transition system declaration or LTL assertion declaration

4.3.5.2 Type declarations

The following inference rules define the translation operator $\rhd$ for type declarations:

$$\frac{type\_def_1,\ ..\ ,\ type\_def_n \rhd type\_def'_1,\ ..\ ,\ type\_def'_n}{\begin{array}{l} st \vdash \textbf{type} \\ \qquad type\_def_1, \\ \qquad .. \\ \qquad type\_def_n \\ \rhd \\ type\_def'_1\ ..\ type\_def'_n,\ \{\} \end{array}} \tag{4.6}$$

where *type_ def$_i$* is an abbreviation type definition or a variant type definition

$$\frac{c \neq st}{c \vdash type\_decl \rhd \epsilon} \tag{4.7}$$

where *type_ decl* is a type declaration

4.3.5.3 Value declarations

The following inference rules define the translation operator $\rhd$ for value declarations:

$$\frac{value\_def_1,\ ..\ ,\ value\_def_n \rhd value\_def'_1,\ ..\ ,\ value\_def'_n}{\begin{array}{l} st \vdash \textbf{value} \\ \qquad value\_def_1, \\ \qquad .. \\ \qquad value\_def_n \\ \rhd \\ value\_def'_1\ ..\ value\_def'_n,\ \{\} \end{array}} \tag{4.8}$$

where *value_ def$_i$* is an explicit value definition or an explicit function definition

$$\frac{c \neq st}{c \vdash value\_decl \rhd \epsilon} \tag{4.9}$$

where *value_ decl* is a value declaration

4.3.5.4 Transition system declarations

The following inference rule defines the translation operator $\triangleright$ for transition system declarations:

$$\frac{st \vdash var\_def_1,\ ..,\ var\_def_n \triangleright var\_def'_1,\ vs_1,\ ..,\ var\_def'_n,\ vs_n}{st \vdash \textbf{transition\_system}[s]} \tag{4.10}$$

$$\begin{array}{l}
\quad\textbf{local} \\
\quad\ \ var\_def_1, \\
\quad\ \ .. \\
\quad\ \ var\_def_n \\
\quad\textbf{in} \\
\quad\ \ trans\_rule_1 \\
\quad\ \ [] \\
\quad\ \ .. \\
\quad\ \ [] \\
\quad\ \ trans\_rule_n \\
\quad\textbf{end} \\
\triangleright \\
var\_def'_1\ ..\ var\_def'_n,\ vs_1 \cup .. \cup vs_n
\end{array}$$

where *var_ def$_i$* is a variable definition
and *trans_ rule$_i$* is a transition rule definition

$$\frac{iv \vdash var\_def_1,\ ..\ ,\ var\_def_n \rhd var\_def'_1,\ ..\ ,\ var\_def'_n}{\begin{array}{l} iv \vdash \textbf{transition\_system}[s] \\ \quad \textbf{local} \\ \quad\quad var\_def_1, \\ \quad\quad .. \\ \quad\quad var\_def_n \\ \quad \textbf{in} \\ \quad\quad trans\_rule_1 \\ \quad\quad [] \\ \quad\quad .. \\ \quad\quad [] \\ \quad\quad trans\_rule_n \\ \quad \textbf{end} \\ \rhd \\ var\_def'_1\ ..\ var\_def'_n \end{array}} \tag{4.11}$$

where var_def_i is a variable definition
and $trans\_rule_i$ is a transition rule definition

$$\frac{tr,\ vs \vdash trans\_rule_1,\ ..\ ,\ trans\_rule_n \rhd trans\_rule'_1,\ ..\ ,\ trans\_rule'_n}{\begin{array}{l} tr,\ vs \vdash \textbf{transition\_system}[s] \\ \quad \textbf{local} \\ \quad\quad var\_def_1, \\ \quad\quad .. \\ \quad\quad var\_def_n \\ \quad \textbf{in} \\ \quad\quad trans\_rule_1 \\ \quad\quad [] \\ \quad\quad .. \\ \quad\quad [] \\ \quad\quad trans\_rule_n \\ \quad \textbf{end} \\ \rhd \\ trans\_rule'_1 \mid\mid .. \mid\mid trans\_rule'_n \end{array}} \tag{4.12}$$

where *var_ def* is a variable definition
and *trans_ rule* is a transition rule definition

$$\frac{c = ps}{c \vdash trans\_sys\_decl \rhd \epsilon} \tag{4.13}$$

where *trans_ sys_ decl* is a type declaration

4.3.5.5 LTL assertion declarations

The following inference rules define the translation operator $\rhd$ for LTL assertion declarations:

$$\frac{ps \vdash ltl\_def_1, \,..\,, \, ltl\_def_n \rhd ltl\_def'_1, \,..\,, \, ltl\_def'_n}{\begin{array}{l} ps \vdash \textbf{ltl\_assertion} \\ \qquad ltl\_def_1, \\ \qquad .. \\ \qquad ltl\_def_n \\ \rhd \\ ltl\_def'_1 \,..\, ltl\_def'_n \end{array}} \tag{4.14}$$

where *ltl_ def$_i$* is a ltl assertion definition

$$\frac{c \neq ps}{c \vdash ltl\_decl \rhd \epsilon} \tag{4.15}$$

where *ltl_ decl* is a LTL assertion declaration declaration

4.3.5.6 Abbreviation type definitions

The following inference rule defines the translation operator $\rhd$ for abbreviation type definitions:

$$\frac{type\_expr \rhd type\_expr'}{id = type\_expr \;\rhd\; id == type\_expr'} \tag{4.16}$$

where *type_ expr* is a type name, type literal or a subtype expression

4.3.5.7 Variant type definitions

The following inference rule defines the translation operator $\rhd$ for variant type definitions:

$$\frac{e_1,\ ..\ ,\ e_n \rhd e'_1,\ ..\ ,\ e'_n}{s == e_1 \,|\, ..\, |\, e_n \ \rhd\ s == e'_1 \,|\, ..\, |\, e'_n} \tag{4.17}$$

4.3.5.8 Explicit value definitions

The following inference rule defines the translation operator $\rhd$ for explicit value definitions:

$$\frac{type \rhd type',\ e \rhd e'}{s\ :\ type = e\ \rhd\ const\ type'\ s == e'} \tag{4.18}$$

4.3.5.9 Explicit function definitions

The following inference rule defines the translation operator $\rhd$ for explicit function definitions:

$$\frac{e_1,\ ..\ ,\ e_n \rhd e'_1,\ ..\ ,\ e'_n,\ type,\ ..\ ,\ type_n \rhd type',\ ..\ ,\ type'_n,\ ve \rhd ve'}{\begin{array}{l} id : type_1 >< \ ..\ >< type_n \to type \\ id(e_1,\ ..\ ,\ e_n)\ is\ ve \\ \rhd \\ type'\ id(type'_1\ e'_1,\ ..\ ,type'_n\ e'_n)\ \{return\ ve'\} \end{array}} \tag{4.19}$$

4.3.5.10 Variable definitions

The following inference rules define the translation operator $\rhd$ for variable definitions:

$$\frac{type \rhd type'}{st \vdash s : type := e\ \rhd\ s\ type',\ \{s\}} \tag{4.20}$$

$$\frac{e \rhd e'}{iv \vdash s : type := e\ \rhd\ s == e'} \tag{4.21}$$

4.3.5.11 Transition rule definitions

The following inference rules define the translation operator $\rhd$ for transition rule definitions:

Here all declared variables are updated in the transition rule

$$\frac{vs \setminus \{id_1,\, ..\,,\, id_n\} = \emptyset,\; be \rhd be',\; ve_1,\, ..\,,\, ve_n \rhd ve'_1,\, ..\,,\, ve'_n}{\begin{array}{l} vs \vdash be \longrightarrow id'_1 = ve_1,\, ..\,,\, id'_n = ve_n \\ \rhd \\ (be' \;\&\&\; id'_1 == ve'_1 \;\&\&\; ..\; \&\&\; id'_n == ve'_n) \end{array}} \tag{4.22}$$

Here not all declared variables are updated in the transition rule

$$\frac{\begin{array}{c} vs \setminus \{id_1,\, ..\,,\, id_n\} = \{uc_1,\, ..\,,\, uc_m\}, \\ be \rhd be',\; ve_1,\, ..\,,\, ve_n \rhd ve'_1,\, ..\,,\, ve'_n \end{array}}{\begin{array}{l} vs \vdash be \longrightarrow id'_1 = ve_1,\, ..\,,\, id'_n = ve_n \\ \rhd \\ (be' \;\&\&\; id'_1 == ve'_1 \;\&\&\; ..\; \&\&\; id'_n == ve'_n \\ \&\&\; uc'_1 == uc_1 \;\&\&\; ..\; \&\&\; uc'_m == uc_m) \end{array}} \tag{4.23}$$

4.3.5.12 LTL assertion definitions

The following inference rules define the translation operator $\rhd$ for LTL assertion definitions:

$$\frac{be \rhd be'}{G(be) \;\rhd\; Globally[be']} \tag{4.24}$$

$$\frac{be \rhd be'}{F(be) \;\rhd\; Finally[be']} \tag{4.25}$$

$$\frac{be \rhd be'}{X(be) \;\rhd\; Next[be']} \tag{4.26}$$

$$\frac{be_1 \rhd be'_1,\; be_2 \rhd be'_2}{U(be_1,\, be_2) \;\rhd\; [be'_1]Until[be'_2]} \tag{4.27}$$

4.3.5.13 Identifiers

The following inference rule defines the translation operator $\rhd$ for identifiers:

$$\frac{}{e \ \rhd \ e} \tag{4.28}$$

where e is an identifier

4.3.5.14 Operators

The following inference rules define the translation operator $\rhd$ for operators. Note that the operator used in the intermediate language is different in some cases, such as $\wedge$ becoming &&. This is such that the intermediate language matches the existing *toString* methods and parsers used in RT-Tester.

$$\frac{}{= \rhd ==} \tag{4.29}$$

$$\frac{}{+ \rhd +} \tag{4.30}$$

$$\frac{}{- \rhd -} \tag{4.31}$$

$$\frac{}{* \rhd *} \tag{4.32}$$

$$\frac{}{/ \rhd /} \tag{4.33}$$

$$\frac{}{< \rhd <} \tag{4.34}$$

$$\frac{}{> \rhd >} \tag{4.35}$$

$$\frac{}{\leq \rhd \leq} \tag{4.36}$$

$$\frac{}{\geq \rhd \geq} \tag{4.37}$$

$$\frac{}{\vee \rhd ||} \tag{4.38}$$

$$\frac{}{\wedge \rhd \&\&} \tag{4.39}$$

$$\frac{}{\sim \rhd !} \tag{4.40}$$

4.3.5.15 Type names

The following inference rule defines the translation operator $\triangleright$ for type names:

$$\frac{}{\mathit{type} \triangleright \mathit{type}} \tag{4.41}$$

4.3.5.16 Type literals

The following inference rules define the translation operator $\triangleright$ for type literals:

$$\frac{}{\textbf{Int} \ \triangleright \ \mathit{int}} \tag{4.42}$$

$$\frac{}{\textbf{Real} \ \triangleright \ \mathit{real}} \tag{4.43}$$

$$\frac{}{\textbf{Bool} \ \triangleright \ \mathit{bool}} \tag{4.44}$$

4.3.5.17 Subtype expressions

The following inference rule defines the translation operator $\triangleright$ for subtype expressions:

$$\frac{\mathit{type} \triangleright \mathit{type}',\ \mathit{ve} \triangleright \mathit{ve}'}{s_1 = \{|\ s_2\ :\ \mathit{type} : -\mathit{ve}\ |\} \ \triangleright \ s_1 == \mathit{type}'\ s_2\ \mathit{where}\ \mathit{ve}'} \tag{4.45}$$

4.3.5.18 Variable names

The following inference rule defines the translation operator $\triangleright$ for variable names:

$$\frac{}{\mathit{id} \triangleright \mathit{id}} \tag{4.46}$$

4.3.5.19 Value names

The following inference rule defines the translation operator $\triangleright$ for value names:

$$\frac{}{\mathit{id} \triangleright \mathit{id}} \tag{4.47}$$

4.3.5.20 Value literals

The following inference rule defines the translation operator $\rhd$ for value literals:

$$\frac{}{e \ \rhd \ e} \tag{4.48}$$

where e is a value literal

4.3.5.21 Function application expressions

The following inference rule defines the translation operator $\rhd$ for function application expressions:

$$\frac{ve_1,\ ..\ ,\ ve_n \rhd ve'_1,\ ..\ ,\ ve'_n}{id(ve_1,\ ..\ ,\ ve_n) \rhd id(ve'_1,\ ..\ ,\ ve'_n)} \tag{4.49}$$

4.3.5.22 Value infix expressions

The following inference rules define the translation operator $\rhd$ for value infix expressions:

$$\frac{ve_1 \rhd ve'_1,\ ve_2 \rhd ve'_2,\ op \rhd op'}{ve_1\ op\ ve_2 \ \rhd\ ve'_1\ op'\ ve'_2} \tag{4.50}$$

where ve_1 and ve_2 are in normal form

$$\frac{\begin{array}{l} ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2, \\ rw\_ve_1\ op\ rw\_ve_2 \equiv rw\_infix, \\ rw\_infix \rhd rw\_infix' \end{array}}{ve_1\ op\ ve_2 \ \rhd\ rw\_infix'} \tag{4.51}$$

4.3.5.23 Value prefix expressions

The following inference rules define the translation operator $\rhd$ for value prefix expressions:

$$\frac{ve \rhd ve'}{\sim ve \rhd\ !ve'} \tag{4.52}$$

where *ve* is in normal form

$$\frac{ve \equiv rw\_ve,\ \sim rw\_ve \equiv rw\_prefix,\ rw\_prefix \rhd rw\_prefix'}{\sim ve\ \rhd\ rw\_prefix'} \tag{4.53}$$

4.3.5.24 If expressions

The following inference rules define the translation operator $\rhd$ for if expressions:

If expression with elsif-branches

$$\frac{\begin{array}{l}\textbf{if } be \textbf{ then } ve_1\ elsif\_branch \textbf{ else } ve_2 \textbf{ end} \equiv rw\_if\_expr,\\ rw\_if\_expr \rhd if\_expr'\end{array}}{\textbf{if } be \textbf{ then } ve_1\ elsif\_branch \textbf{ else } ve_2 \textbf{ end} \rhd if\_expr'} \tag{4.54}$$

where *elsif_ branch* is an elsif branch and
where *ve₁* and *ve₂* are boolean expressions

If expression without elsif-branches

$$\frac{be \rhd be',\ ve_1 \rhd ve_1',\ ve_2 \rhd ve_2'}{\begin{array}{l}\textbf{if } be \textbf{ then } ve_1 \textbf{ else } ve_2 \textbf{ end}\\ \rhd\\ ((be' \mathbin{\&\&} ve_1') \mathbin{||} (!(be') \mathbin{\&\&} ve_2'))\end{array}} \tag{4.55}$$

where *ve₁* and *ve₂* are boolean expressions and
where *be*, *ve₁* and *ve₂* are in normal form

$$\frac{\begin{array}{l}be \equiv rw\_be,\ ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2,\\ \textbf{if } rw\_be \textbf{ then } rw\_ve_1 \textbf{ else } rw\_ve_2 \rhd rw\_if\_expr' \textbf{ end}\end{array}}{\textbf{if } be \textbf{ then } ve_1 \textbf{ else } ve_2 \textbf{ end} \rhd rw\_if\_expr'} \tag{4.56}$$

where *ve₁* and *ve₂* are boolean expressions

4.3.5.25 Case expressions

Note that case expressions are not translated directly, but rather rewritten into an equivalent if expression and then translated.

The following inference rule defines the translation operator $\triangleright$ for case expressions:

$$\frac{case\_expr \equiv if\_expr,\ if\_expr \triangleright if\_expr'}{case\_expr \triangleright if\_expr'} \tag{4.57}$$

where *case_ expr* is a case expression

4.3.5.26 Rewriting value expressions

The following inference rules define the rewriting operator $\equiv$ for value expressions.
The inference rules below should be applied in the order they are presented, such that the first rule has the highest priority. An example showing the use of these rules can be found in Appendix A.2.

Rewrite case expression into equivalent if expression

$$\frac{ve_1,\ ..\ , ve_{n+1} \equiv rw\_ve_1,\ ..\ , rw\_ve_{n+1}}{\begin{array}{l} \text{case } e \text{ of} \\ \quad e_1 \rightarrow ve_1 \\ \quad e_2 \rightarrow ve_2 \\ \quad .. \\ \quad e_n \rightarrow ve_n \\ \quad \_ \rightarrow ve_{n+1} \\ \equiv \\ \textbf{if } e = e_1 \textbf{ then } rw\_ve_1 \textbf{ else} \\ \quad \textbf{if } e = e_2 \textbf{ then } rw\_ve_2 \textbf{ else} \\ \quad\quad .. \\ \quad\quad\quad \textbf{if } e = e_n \textbf{ then } rw\_ve_n \textbf{ else } rw\_ve_{n+1} \textbf{ end} \\ \quad\quad .. \\ \quad \textbf{end} \\ \textbf{end} \end{array}} \tag{4.58}$$

Rewrite if expression containing else if branches

$$\frac{\begin{array}{l} ve \equiv rw\_ve,\ ve_1,\ ..\ ,ve_n \equiv rw\_ve_1,\ ..\ ,rw\_ve_n,\\ be_1,\ ..\ ,be_n \equiv rw\_be_1,\ ..\ ,rw\_be_n \end{array}}{\begin{array}{l} \textbf{if}\ be_1\ \textbf{then}\ ve_1\\ \quad \textbf{elsif}\ be_2\ \textbf{then}\ ve_2\\ \quad ..\\ \quad \textbf{elsif}\ be_n\ \textbf{then}\ ve_n\\ \textbf{else}\ ve\ \textbf{end}\\ \equiv\\ \textbf{if}\ rw\_be_1\ \textbf{then}\ rw\_ve_1\ \textbf{else}\\ \quad \textbf{if}\ rw\_be_2\ \textbf{then}\ rw\_ve_2\ \textbf{else}\\ \quad\quad ..\\ \quad\quad\quad \textbf{if}\ rw\_be_n\ \textbf{then}\ rw\_ve_n\ \textbf{else}\ rw\_ve\\ \quad\quad ..\\ \quad \textbf{end}\\ \textbf{end} \end{array}} \tag{4.59}$$

Rewrite two if expressions used in the same value expression

$$\frac{\begin{array}{l} ve_1 \equiv rw\_ve_1,\ ve'_1 \equiv rw\_ve'_1,\ ve_2 \equiv rw\_ve_2,\ ve'_2 \equiv rw\_ve'_2,\\ be_1 \equiv rw\_be_1,\ be_2 \equiv rw\_be_2\\ rw\_ve_1\ op\ rw\_ve_2 \equiv rw\_infix_1,\ rw\_ve_1\ op\ rw\_ve'_2 \equiv rw\_infix_2,\\ rw\_ve'_1\ op\ rw\_ve_2 \equiv rw\_infix_3,\ rw\_ve'_1\ op\ rw\_ve'_2 \equiv rw\_infix_4 \end{array}}{\begin{array}{l} \textbf{if}\ be_1\ \textbf{then}\ ve_1\ \textbf{else}\ ve'_1\ \textbf{end}\\ op\\ \textbf{if}\ be_2\ \textbf{then}\ ve_2\ \textbf{else}\ ve'_2\ \textbf{end}\\ \equiv\\ \textbf{if}\ rw\_be_1 \wedge rw\_be_2\ \textbf{then}\ rw\_infix_1\ \textbf{else}\\ \quad \textbf{if}\ rw\_be_1 \wedge \sim rw\_be_2\ \textbf{then}\ rw\_infix_2\ \textbf{else}\\ \quad\quad \textbf{if}\ \sim rw\_be_1 \wedge rw\_be_2\ \textbf{then}\ rw\_infix_3\ \textbf{else}\ rw\_infix_4\\ \quad\quad \textbf{end}\\ \quad \textbf{end}\\ \textbf{end} \end{array}} \tag{4.60}$$

Rewrite if expression used in infix value expression (right)

$$\frac{be \equiv rw\_be,\ ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2 \quad e\ op\ rw\_ve_1 \equiv rw\_infix_1,\ e\ op\ rw\_ve_2 \equiv rw\_infix_2}{\begin{array}{l} e\ op\ \textbf{if}\ be\ \textbf{then}\ ve_1\ \textbf{else}\ ve_2\ \textbf{end} \\ \equiv \\ \textbf{if}\ rw\_be\ \textbf{then}\ rw\_infix_1\ \textbf{else}\ rw\_infix_2\ \textbf{end} \end{array}} \tag{4.61}$$

Rewrite if expression used in infix value expression (left)

$$\frac{be \equiv rw\_be,\ ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2 \quad rw\_ve_1\ op\ e \equiv rw\_infix_1,\ rw\_ve_2\ op\ e \equiv rw\_infix_2}{\begin{array}{l} \textbf{if}\ be\ \textbf{then}\ ve_1\ \textbf{else}\ ve_2\ \textbf{end}\ op\ e \\ \equiv \\ \textbf{if}\ rw\_be\ \textbf{then}\ rw\_infix_1\ \textbf{else}\ rw\_infix_2\ \textbf{end} \end{array}} \tag{4.62}$$

Rewrite if expression used in prefix value expression

$$\frac{be \equiv rw\_be,\ ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2}{\begin{array}{l} \sim (\textbf{if}\ be\ \textbf{then}\ ve_1\ \textbf{else}\ ve_2\ \textbf{end}) \\ \equiv \\ \textbf{if}\ rw\_be\ \textbf{then}\ \sim (rw\_ve_1)\ \textbf{else}\ \sim (rw\_ve_2)\ \textbf{end} \end{array}} \tag{4.63}$$

No if expressions to rewrite

$$\frac{}{ve \equiv ve} \tag{4.64}$$

where ve is in normal form

$$\frac{ve_1 \equiv rw\_ve_1,\ ve_2 \equiv rw\_ve_2,\ rw\_ve_1\ op\ rw\_ve_2 \equiv rw\_infix}{ve_1\ op\ ve_2 \equiv rw\_infix} \tag{4.65}$$

4.4 Parsing the intermediate language

The intermediate language is designed in such a way, that it is essentially a string representation of its corresponding RT-Tester model, which makes the parsing process very straightforward. The main challenge is to know which RT-Tester model object to use for each part, which requires some familiarity with the existing source code. However, this is an implementation issue, and does not involve any design choices.

As mentioned in the previous chapter, there already exists a parser for LTL formulae in RT-Tester, which takes a string representation of LTL formulae and creates the corresponding RT-Tester model objects. This parser is reused in this project, and the contents of the property specification part of the intermediate language model is designed to work with the parser without any changes.

The existing LTL parser also makes the parsing of the initial values and the transition relation easier. Both of these are simple logical expressions, so the part of the LTL parser, which reads logical expressions can also be reused here.

The biggest task is to create the symbol table based on the content of the intermediate language model. The existing parser cannot be used here, so this part needs to be implemented from scratch.

4.5 Variable bounds

It is a requirement when doing model checking in RT-Tester, that the variables within the model are bounded. This means that all variables must have an associated finite range of possible values, which are allowed. The reason for this requirement is, that the underlying SMT solver, which RT-Tester uses to perform model checking, works by assigning values to all variables, such that a given logical formula is satisfied. If there is an infinite set of possible values, the SMT solver can try different assignments of values forever, without a clear way of knowing if the logical formula is satisfiable.

The idea of bounded variables presents an issue when translating a model specified in RSL to the corresponding model in RT-Tester, since RSL allows variables to be unbounded. Because of this, any time an RSL model which contains unbounded variables (such as a simple integer variable) is represented in RT-Tester, the state space of the corresponding RT-Tester model will be smaller compared to the RSL model.

At first glance, this seems like a major problem, since a translated RT-Tester model ideally should represent the same state space as the original RSL model. But in reality, it is only the reachable state space that must remain the same. Another way of looking at this requirement, is that the model checking performed on the translated model, should yield the same result as for the original model. If we imagine that all of the missing states from the state space of the translated model are actually unreachable states, then the model checking result will remain the same, despite the smaller state space.

Only eliminating unreachable states is not always possible to accomplish. Consider the following RSL specification:

```
scheme infinite_state_space =
  class
    transition_system [T]
      local
        x : Int := 1
      in
        [incrementX] x > 0 ⟶ x′ = x + 1
      end
  end
```

This model has an infinite reachable state space. Any translated model will have a finite state space, due to the requirement of variable ranges. Therefore it cannot always be avoided, that the translated model will have a smaller reachable state space.
However, for RSL specification with a finite reachable state space, it is possible to define the variable bounds such that the reachable state space is identical for the translated model.

The way the variable bounds are handled in this project, is by having a relatively small default range for each variable type. If that range turns out to be insufficient, the user can change the RSL specification, and use *subtype expressions* to define custom type ranges. In terms of identifying whether variable ranges are insufficient, the generation of confidence conditions (available in the rsltc tool) can be used to aid the user.
The idea behind this solution is, that a relatively small variable range will be sufficient in most cases, while keeping the performance impact on the SMT solver negligible. In the few cases where the default ranges negatively impact the reachable state space of the translated model, the user can then provide custom ranged based on the nature of the particular model.
The default ranges for each type has been chosen as follows:

- **Int:** [-100 .. 100]
- **Real:** [-100 .. 100]

- **Bool:** [0 .. 1] - this range is trivial, but still needs to be defined in RT-Tester.

4.6 Providing bounds for model checking

As mentioned in Section 2.3.1, RT-Tester only performs *bounded* model checking, i.e. the model is only checked for a finite number of transition steps. The exact number of steps being checked, can be customized in RT-Tester for each run of the SMT solver.

The way this is being handled, is by allowing the user to give the desired number of steps as an argument to the parser in RT-Tester. If no argument is given, the default number of steps is 10.

Chapter 5

Implementation

In this chapter the implementation of the software developed in this thesis will be outlined. The implementation will be described in three parts:

- The RSL translator, which is an extension of the existing rsltc tool.
- The intermediate language parser, which is implemented using Lex and Yacc.
- The executable in RT-Tester, which performs the model checking based on a parsed model.

The source code for the implementation can be found in the associated *thesis.zip* file.

5.1 RSL translator

The RSL translator is implemented by reusing as much of the existing rsltc tool as possible.
Before the RSL translation is attempted, the existing type checker is used to check the static correctness of the given RSL specification. While the type

checking is performed, an abstract syntax tree of the given specification is build. It is based on this abstract syntax tree that the translation is performed.

5.1.1 Changes to existing files

A few changes have been made to the existing files grammar.g, ext.g, main.c and files.c, which can be found the *rsltc/src* directory. These changes simply to allow the user to access the RSL translator directly in the rsltc executable, using the following command:

```
./rsltc -rtt <filename>.rsl
```

For more on how to use the translator, see Section 8.1.2.

5.1.2 rtt.g

The implementation of the RSL translator is located in the file rtt.g, which can be found the *rsltc/src* directory.

The structure of the implementation is fairly similar to the structure of the system of inference rules defined in Section 4.3.5. It mainly consists of a number of *Actions*, which is a Gentle construction. Many of the Actions in this file have the prefix 'gen' (e.g. gen_class). These Actions are responsible for generating the intermediate language code.

Most of the generating Actions have an associated type in the abstract syntax tree that they generate code for. An example of this, is the gen_class Action which translates the type CLASS as seen below:

```
'type' CLASS

    basic          (decls     :   DECLS)
    extend         (left      :   CLASS,
                    right     :   CLASS)
    hide           (hides     :   DEFINEDS,
                    class     :   CLASS)
    rename         (renames   :   RENAMES,
                    class     :   CLASS)
    with           (pos       :   POS,
                   objects  :   OBJECT_EXPRS,
                   class    :   CLASS)
     instantiation (name    :   NAME,
                   parm     :   OBJECT_EXPRS)
     nil


 'action' gen_class(CLASS)

   'rule' gen_class(basic(Ds)):
          check_enums_for_duplicates(Ds, nil)
          WriteFile("SYM_TABLE_DECL")
          gen_sym_table_decls(Ds)
          WriteFile("\nSYM_TABLE_DECL_END\n\nINIT_VAL")
          gen_init_values(Ds)
          WriteFile("\nINIT_VAL_END\n\nTRANS_REL")
          gen_transition_relations(Ds)
          WriteFile("\nTRANS_REL_END\n\nPROP_SPEC")
          gen_property_specifications(Ds)
          WriteFile("\nPROP_SPEC_END")
   'rule' gen_class(_):
          ErrorUsage("Error: only basic classes are accepted")
```

In general, the rules of an action determine which instances of a given type is accepted by the translator. In this example, the only instance of type CLASS that is accepted is basic(decls : DECLS). For any other instance, the program is aborted and an error message is printed.

In the system of inference rules, a *context* is also provided for rules which should only be used in a certain context. The same principle is also used in the implementation of the translator. The following type CONTEXT is defined:

```
 'type' CONTEXT

    sym_table_decl
    init_val
    trans_rel
```

This type is used as parameter for certain Actions, where they are relevant to the code generation. An example of this, is the *gen_ variable_ def* Action, responsible for generating the code for variable definitions:

```
'action' gen_variable_def(VARIABLE_DEF, CONTEXT)

  'rule' gen_variable_def(single(_, Id, Type, _),
             sym_table_decl):
          WriteFile("\n")
          gen_type_expr(Type)
          WriteFile(" ")
          id_to_string(Id -> S)
          WriteFile(S)

  'rule' gen_variable_def(single(_, Id, _, initial(VE)),
             init_val):
          WriteFile("\n")
          id_to_string(Id -> S)
          WriteFile(S)
          WriteFile(" == ")
          gen_value_expr(VE)
```

Here the two rules defined in *gen_ variable_ def* correspond to the two inference rules (4.20) and (4.21) respectively, and the context keys *sym_ table_ decl* and *init_ val* is used to distinguish the two.
However, there are some cases where separate Actions are created rather than using the CONTEXT type. This can be seen in the way the type DECLS is handled, where four different Actions are used. Here it is the Rules within each Action that determines whether a certain declaration is translated or not, rather than a CONTEXT:

```
'action' gen_sym_table_decls(DECLS)
  ...
  'rule' gen_sym_table_decls(list(type_decl(_, TDs), Ds)):
          gen_type_declarations(TDs)
          gen_sym_table_decls(Ds)

  'rule' gen_sym_table_decls(list(value_decl(_, VDs), Ds)):
          gen_value_declarations(VDs)
          gen_sym_table_decls(Ds)

  'rule' gen_sym_table_decls(list(trans_system_decl(_, TSs),
             Ds)):
          gen_transition_systems(TSs, sym_table_decl)
          gen_sym_table_decls(Ds)
  ...
```

```
'action' gen_init_values(DECLS)
  ...
  'rule' gen_init_values(list(trans_system_decl(_, TSs), Ds)):
           gen_transition_systems(TSs, init_val)
           gen_init_values(Ds)
  ...

'action' gen_transition_relations(DECLS)
  ...
  'rule' gen_transition_relations(list
               (trans_system_decl(_, TSs), Ds)):
           gen_transition_systems(TSs, trans_rel)
  ...

'action' gen_property_specifications(DECLS)
  ...
  'rule' gen_property_specifications(list(property_decl(_, P),
               Ds)):
           gen_properties(P)
  ...
```

This example does not correspond directly to the system of inference rules, however it will still generate code identical to the intended design. The reason for this inconsistency between the design and implementation, is that the system of inference rules was developed after some parts of the implementation.

The code being generated in the different Actions is written string by string directly to an output file. If the file being translated is *example.rsl*, the generated intermediate language code will be written to a file *example.rtt* in the same folder as the executable.

5.1.2.1 Rewriting if- and case expressions

In the system of inference rules defined in Section 4.3.5, any value expression, which is not in normal form, is rewritten according to the equivalence operator $\equiv$ as defined in Section 4.3.5.26.
In the implementation of the translator, the similar Condition *has_if_expr* is used to check whether a given value expression contains any if or case expressions. If a value expression contains an if or case expression, the Action *rewrite_if_expr* is used:

```
'action' rewrite_if_expr(VALUE_EXPR -> VALUE_EXPR)

  'rule' rewrite_if_expr(if_expr(P, G, T, P2, nil, else(P3, E))
           -> if_expr(P, G2, T2, P2, nil, else(P3, E2))):
           ...

  'rule' rewrite_if_expr(if_expr(P, G, T, P2, EIs, else(P3, E))
           -> NewIf):
           ...

  'rule' rewrite_if_expr(val_infix(_,
           if_expr(Pl, Gl, Tl, P2l, EIsl, El), Op,
           if_expr(Pr, Gr, Tr, P2r, EIsr, Er)) -> NewIf):
           ...

  'rule' rewrite_if_expr(val_infix(_, Left, Op,
           if_expr(P, G, T, P2, EIs, else(P4, E))) -> NewIf):
           ...

  'rule' rewrite_if_expr(val_infix(_,
           if_expr(P, G, T, P2, EIs, else(P4, E)), Op, Right)
           -> NewIf):
           ...

  'rule' rewrite_if_expr(val_infix(P, Left, Op, Right)
           -> NewIf):
           ...

  'rule' rewrite_if_expr(ax_prefix(P, not, VE)
           -> ax_prefix(P, not, VE2)):
           ...

  'rule' rewrite_if_expr(case_expr(P, VE, P2, Bs) -> IF):
           ...

  'rule' rewrite_if_expr(VE -> VE):
           ...
```

The Rules used in this Action are similar to the inference rules defined in Section 4.3.5.26.

The first Rule corresponds to inference rule (4.56), dealing with the rewriting of if expressions such that their nested value expressions also are rewritten.

The second Rule corresponds to inference rule (4.59), dealing with if expression containing else-if branches.

The third Rule corresponds to inference rule (4.60), dealing with infix expressions where both sides are if expressions.

The fourth Rule corresponds to inference rule (4.61), dealing with infix expressions where only the left side is an if expression.

The fifth Rule corresponds to inference rule (4.62), dealing with infix expressions where only the right side is an if expression.

The sixth Rule corresponds to inference rule (4.63), dealing with prefix expressions.

The seventh Rule corresponds to inference rules (4.64) and (4.65), dealing with infix expressions where neither side is an if expression.

The eighth Rule corresponds to inference rule (4.58), dealing with case expressions.

The last Rule corresponds to inference rule (4.65), where none of the other cases matches and nothing is rewritten.

5.2 Intermediate language parser

The intermediate language parser is implemented in the files rttparser_lex.ll and rttparser_yacc.ypp, which are located in the *rtttgen_ V2/parsers/rttparser* directory. These files are based on the existing LTL parser files ltlparser_lex.ll and ltlparser_yacc.ypp, which are located in the *rtttgen_ V2/parsers/ltlparser* directory.

Some auxiliary classes which are used by the parser have also been created: The files RttParserOutput.cpp, RttParserOutput.h, RttSubtype.cpp and RttSubtype.h, all located in the *rtttgen_ V2/rsl* directory.

5.2.1 RttParserOutput.cpp/h

This class is simply a wrapper class for the different output types the parser produces.
The parser produces five different outputs: A map from ids to subtypes, a symbol table, an initial value constraint, a transition relation and a list of property specifications.

```
class RttTgenParserOutput{

private:
    map<string,RttSubtype*> *subtypeMap;
    RttTgenSymbolTable *symbolTable;
    RttTgenExpTree *initialValues;
    RttTgenExpTree *transitionRelation;
    vector<RttTgenLTLFormula*> *propertySpecifications;

...
}
```

5.2.2 RttSubtype.cpp/h

This class is used to store relevant information about a single subtype defined in the parsed model.
In each instance of this class, the following is stored: The name of the subtype, its maximal type, the value expression which constrains the subtype, and the name of the local variable used in the value expression.

```
class RttSubtype{

private:
    string *typeName;
    string *subtypeOf;
    string *localVarName;
    RttTgenExpTree *valueExpression;

...
}
```

5.2.3 rttparser_lex.ll

By reusing the file ltlparser_lex.ll, this lexer is already capable of producing most of the tokens needed to parse the intermediate language. Only the following was added to this file:

```
"SYM_TABLE_DECL"          { rttparser_currentline +=
   rttparser_text; return(SYM_TABLE_DECL_START); }
"SYM_TABLE_DECL_END"      { rttparser_currentline +=
   rttparser_text; return(SYM_TABLE_DECL_END); }
"INIT_VAL"                { rttparser_currentline +=
   rttparser_text; return(INIT_VAL_START); }
"INIT_VAL_END"            { rttparser_currentline +=
   rttparser_text; return(INIT_VAL_END); }
"TRANS_REL"              { rttparser_currentline +=
   rttparser_text; return(TRANS_REL_START); }
"TRANS_REL_END"          { rttparser_currentline +=
   rttparser_text; return(TRANS_REL_END); }
"PROP_SPEC"              { rttparser_currentline +=
   rttparser_text; return(PROP_SPEC_START); }
"PROP_SPEC_END"          { rttparser_currentline +=
   rttparser_text; return(PROP_SPEC_END); }

"const"                  { rttparser_currentline +=
   rttparser_text; return(CONST); }
"return"                 { rttparser_currentline +=
   rttparser_text; return(RETURNKEY); }
"where"                  { rttparser_currentline +=
   rttparser_text; return(WHEREKEY); }
"'"              { rttparser_currentline +=
   rttparser_text; return(PRIME); }
```

The functions *parse_rtt_string* and *parse_rtt_file* were also changed to return *RttParserOutput* instead of *RttTgenLTLFormula*.

5.2.4 rttparser_yacc.ypp

This file is based on the file ltlparser_yacc.ypp. The existing code has only been changed slightly, but since parsing LTL formulae is only part of what needs to be done, a lot has also been added.

A number of global variables have been added to the parser, which are used throughout the parser:

```
RttTgenParserOutput *parseroutput = new RttTgenParserOutput();
RttTgenSymbolTable *symboltable = new RttTgenSymbolTable(true);
map<string, RttSubtype*> *typeNameToSubtype =
  new map<string, RttSubtype*>();
map<string, RttSubtype*> *varNameToSubtype =
  new map<string, RttSubtype*>();
vector<RttTgenLTLFormula*> *formulae =
  new vector<RttTgenLTLFormula*>();

string int_min = "-100";
string int_max = "100";
string bool_min = "0";
string bool_max = "1";
string real_min = "-100";
string real_max = "100";
```

To accommodate the new tokens added in rttparser_lex.ll, the following token declarations have been added:

```
%token PRIME CONST RETURNKEY WHEREKEY
%token SYM_TABLE_DECL_START SYM_TABLE_DECL_END INIT_VAL_START
       INIT_VAL_END TRANS_REL_START TRANS_REL_END
       PROP_SPEC_START PROP_SPEC_END
```

Finally, a number of rules have been added, enabling the parsing of the symbol table, the initial values of variables and the transition relation.
The starting rule of the parser is *parser_output*, where the content of the RttParserOutput class returned by the parser is set:

```
parser_output:
sym_table_declarations_final initial_values_final
  transition_relation property_specifications_final
{
parseroutput->setSymbolTable(symboltable);
parseroutput->setInitialValues($2);
parseroutput->setTransitionRelation($3);
parseroutput->setPropertySpecifications(formulae);
parseroutput->setSubtypeMap(varNameToSubtype);
}
```

In addition, the following rules have been created to parse the symbol table, initial values of variables and transition relation:

```
sym_table_declarations_final:
SYM_TABLE_DECL_START sym_table_declarations
  SYM_TABLE_DECL_END { }

sym_table_declarations:
sym_table_declaration { }
| sym_table_declarations sym_table_declaration { }

sym_table_declaration:
```

```
variable_declaration { }
| function_declaration { }
| constant_declaration { }
| subtype_declaration { }
| enum_declaration { }

initial_values_final:
INIT_VAL_START initial_values INIT_VAL_END { $$ = $2; }

initial_values:
equality_expression { $$ = $1; }
| initial_values equality_expression
  { $$ = new RttTgenExpTree("&&", INFIXOPERATOR, BOOLAND);
    $$->setLeft($1);
    $$->setRight($2);
$$->setVersion(0,true); }

transition_relation:
TRANS_REL_START logical_or_expression TRANS_REL_END { $$ = $2; }

property_specifications_final:
PROP_SPEC_START property_specifications PROP_SPEC_END {  }

property_specifications:
pathformula {  }
| property_specifications pathformula {  }
```

Note here that the rules *initial_ values*, *transition_ relation* and *property_ specification* are reusing rules from the original LTL parser to create the appropriate objects.

To give an example of how these rules work, consider the following section from the rule *variable_ declaration*:

```
variable_declaration:
INT_T IDENT
{ map<string,RttTgenType*> &typemap = symboltable->getTypeMap();
  RttTgenType *type = typemap["signed long int"];
  RttTgenSymbolTableVarEntry* entry =
    new RttTgenSymbolTableVarEntry(
       *type,
       new RttTgenVariableSymbol(string($2),rttTgenInputVar,
         "signed long int",int_min,int_min,int_max),
       SYMTAB_IMR);
symboltable->addVarEntry(string($2),entry); }
| REAL_T IDENT
{ ... }
| BOOL_T IDENT
{ ... }
| IDENT IDENT
{ ... }
```

As shown here, there are four cases of variable declarations. The variable type is either an int, a real, a bool or a custom type.
In the case of an int, a new variable entry is created using the existing integer type in the symbol table, after which the entry is added to the map of variable entries in the symbol table.

5.2.5 Subtypes

There is not a direct way to represent subtypes in the symbol table. So the way subtypes are handled, is by declaring variables in the symbol table as the maximal type of the subtype. The RttSubtype class is used to store the relevant information about the subtype, such as the constraints being imposed upon the maximal type. The information contained in this class is then used to constrain the values of the subtype, when a global transition relation is created in the main executable (see Section 5.3.2).

This process is best explained with an example. Consider the following subtype declaration and variable declaration:

```
nat == int n where n >= 0
nat x
```

When the subtype declaration is read by the parser, the following RttSubtype class instance is created in the rule *subtype_ declaration*, and added to the global map *typeNameToSubtype*:

```
RttTgenExpTree *constraint = new RttTgenExpTree(...)
RttSubtype *subtype =
  new RttSubtype("nat","int","n",constraint);
typeNameToSubtype->insert(
  pair<string,RttSubtype*>("nat",subtype) );
```

When the variable declaration is then read, the variable is included in the symbol table with the type int, while also being added to the global map *varNameToSubtype*:

```
RttSubtype* subType = (*typeNameToSubtype)["nat"]
varNameToSubtype->insert(
  pair<string,RttSubtype*>("x",subType) );

map<string,RttTgenType*> &typemap = symboltable->getTypeMap();
RttTgenType *type = typemap["int"];
RttTgenSymbolTableVarEntry* entry =
  new RttTgenSymbolTableVarEntry(
    *type,
    new RttTgenVariableSymbol("x",rttTgenInputVar,
      "int",int_min,int_min,int_max),
    SYMTAB_IMR);
symboltable->addVarEntry("x",entry);
```

After the parsing has been completed, the bounds for all variables are added as constraints to the transition relation (as described in Section 5.3.2). Here the map *varNameToSubtype* is also used to include the constraints imposed upon the subtypes. For this particular example, the following constraint will be added to the transition relation:

$$x >= 0$$

where the local variable name n has been substituted with the actual variable name x.

5.3 Model checking

The executable which uses the parser and performs model checking, is implemented in the file rtt-rsl.cpp, located in the *rtttgen_ V2/executables/rtt-rsl* directory.

The file consists of three functions, *main*, *createGlobalTransRel* and *solveGoal*.

5.3.1 main function

The *main* function takes either one or two arguments, the first being the path for the intermediate language file being parsed, and the second being the step bound necessary for model checking. If no second argument is given, the default of 10 steps is used.

The parser is then called to produce an instance of the class *RttParserOutput*. Using the symbol table contained in that class, an instance of the solver class *RttTgenSonolarTransRelSolver* is initialized:

```
RttTgenSonolarTransRelSolver *solver =
  new RttTgenSonolarTransRelSolver(symtab,true);
```

A global transition relation (see Section 5.3.2) is created using the *createGlobalTransRel* function, and is fed to the solver along with the initial values:

```
RttTgenExpTree* globalTrans =
  createGlobalTransRel(transRel,symtab,subtypeMap);
solver->setTransitionRelation(initVals, globalTrans);
```

Finally the list of property specifications (LTL formulae) are model checked one by one using the *solveGoal* function (see Section 5.3.3).

5.3.2 createGlobalTransRel function

This function creates a so-called global transition relation, which is a constraint that must hold in all states of the model checking process (unlike the initial value constraint, which only has to hold in the initial state). The global transition relation is a logical conjunction, consisting of the transition relation coming from the parser, and the bounds imposed on each variable.

The function takes three arguments, a transition relation, a symbol table and a subtype map, and return a global transition relation:

```
RttTgenExpTree* createGlobalTransRel(RttTgenExpTree* transRel,
  RttTgenSymbolTable* symTab, map<string,RttSubtype*>* subtypeMap){
  ...
}
```

Both the variable map contained in the symbol table and the subtype map are then iterated through, adding the variable bound constraints to the transition relation.

One thing to note, is that the symbol table always contains a variable called *_timeTick*, which is used when modelling real time systems. This variable must be skipped when iterating through the variables, and a constraints stating that the variable never changes must also be added to the global transition relation:

```
// add "/\ _timeTick' == _timeTick"
RttTgenExpTree *timeTickExpr =
  new RttTgenExpTree("==", INFIXOPERATOR, COMPEQ);
RttTgenExpTree *timeTickLeft =
  new RttTgenExpTree("_timeTick", IDENTIFIER, NAMEX);
timeTickLeft->setVersion(1,true);
RttTgenExpTree *timeTickRight =
  new RttTgenExpTree("_timeTick", IDENTIFIER, NAMEX);
timeTickRight->setVersion(0,true);
timeTickExpr->setLeft(timeTickLeft);
timeTickExpr->setRight(timeTickRight);

globalTransRel = RttTgenExpTree::createANDExpr(tempTransRel,
                                                timeTickExpr);
```

5.3.3 solveGoal function

The function *solveGoal* takes four arguments, a property specification (or goal), an initial value constraint, an instance of the solver and a step bound.

```
void solveGoal(RttTgenLTLFormula* propSpec,
  RttTgenExpTree* initialConstraints,
  RttTgenSonolarTransRelSolver* solver, int stepBound){
  ...
}
```

Note that the solver only accept vectors of goals, so the goal argument is simply given as a vector with one element. Another option would be to attempt to solve all the goals stated in the original specification at the same time, but then it would not be possible to distinguish which goal failed in the verification.

The solver method *solveConstraintWithTransRelInitial* is then called using the appropriate arguments:

```
vector< ::latticelib::ConcreteLattice<bool> > stepPossibilities;
vector<RttTgenExpTree*> guidingConstraints;
InitialValues_t initVals;
vector<RttTgenLTLFormula*> goalVec;
goalVec.push_back(propSpec);
unsigned int solverStep = 0;
bool solved = false;

solved = solver->solveConstraintWithTransRelInitial(goalVec,
  &initVals, initialConstraints, stepPossibilities,
  guidingConstraints, stepBound, solverStep, NULL);
```

If a witness is found, the values of all variables (except *_timeTick*) are then printed for each step.

Chapter 6

Testing the RSL translator

In this chapter, the implementation of the RSL translator will be tested, to see if it matches the design presented in Chapter 4. Each test consist of three parts, the test input, the expected output and the actual output. The headline of each test indicates whether that test has passed or failed.

Note that only the RSL constructions which directly produce code in the translation will be tested. This means that constructions such as type declarations will not be tested as such, but rather the type definitions contained within the declaration.
There are also many constructions which cannot be tested without being part of larger construction. For instance, a type name cannot appear alone in an RSL specification, but must be part of a type definition or similar. It will therefore be mentioned after each test, which "sub-constructions" also have been indirectly tested.

6.1 Scheme declarations

6.1.1 Test 1 - Passed

Test input

```
scheme test1 =
  class
  end
```

Expected output

```
SYM_TABLE_DECL
SYM_TABLE_DECL_END

INIT_VAL
INIT_VAL_END

TRANS_REL
TRANS_REL_END

PROP_SPEC
PROP_SPEC_END
```

Actual output

```
SYM_TABLE_DECL
SYM_TABLE_DECL_END

INIT_VAL
INIT_VAL_END

TRANS_REL
TRANS_REL_END

PROP_SPEC
PROP_SPEC_END
```

6.2 Abbreviation type definitions

6.2.1 Test 2 - Passed

Test input

```
scheme test2 =
 class
    type MyInt = Int
 end
```

Expected output

```
SYM_TABLE_DECL
MyInt == int
SYM_TABLE_DECL_END
...
```

Actual output

```
SYM_TABLE_DECL
MyInt == int
SYM_TABLE_DECL_END
...
```

This test also shows that the type name *MyInt* and the type literal *Int* were translated as expected.

6.2.2 Test 3 - Passed

Test input

```
scheme test3 =
```

class
  type MySubtype = $\{| \text{x} : \textbf{Int} \bullet 0 \leq \text{x} \wedge \text{x} \leq 10 |\}$
end

Expected output

```
SYM_TABLE_DECL
MySubtype = int x where 0 <= x && x <= 10
SYM_TABLE_DECL_END
...
```

Actual output

```
SYM_TABLE_DECL
MySubtype = int x where 0 <= x && x <= 10
SYM_TABLE_DECL_END
...
```

This test also shows that the value literals 0 and 10 were translated as expected.

6.3 Variant type definitions

6.3.1 Test 4 - Passed

Test input

scheme test4 =
 class
  type Variant == enum1 | enum2 | enum3
 end

Expected output

```
SYM_TABLE_DECL
Variant == enum1 | enum2 | enum3
SYM_TABLE_DECL_END
...
```

Actual output

```
SYM_TABLE_DECL
Variant == enum1 | enum2 | enum3
SYM_TABLE_DECL_END
...
```

6.4 Explicit value definitions

6.4.1 Test 5 - Passed

Test input

```
scheme test5 =
 class
   value x : Real = 1.0
 end
```

Expected output

```
SYM_TABLE_DECL
const x real == 1.0
SYM_TABLE_DECL_END
...
```

Actual output

```
SYM_TABLE_DECL
const x real == 1.0
SYM_TABLE_DECL_END
...
```

This test also shows that the value name x was translated as expected.

6.5 Explicit function definitions

6.5.1 Test 6 - Passed

Test input

```
scheme test6 =
 class
   value
    f : Int × Int → Bool
    f(x,y) ≡ x = y
 end
```

Expected output

```
SYM_TABLE_DECL
bool f (int x, int y) {return x == y}
SYM_TABLE_DECL_END
...
```

Actual output

```
SYM_TABLE_DECL
bool f (int x, int y) {return x == y}
SYM_TABLE_DECL_END
...
```

This test also shows that the value infix expression $x = y$ was translated as expected.

6.6 Variable definitions

6.6.1 Test 7 - Passed

Test input

```
scheme test7 =
 class
   transition_system [TS]
    local
       x : Bool := false
    in
       ...
    end
 end
```

Expected output

```
SYM_TABLE_DECL
bool x
SYM_TABLE_DECL_END

INIT_VAL
x == false
INIT_VAL_END
...
```

Actual output

```
SYM_TABLE_DECL
bool x
SYM_TABLE_DECL_END
```

```
INIT_VAL
x == false
INIT_VAL_END
...
```

This test also shows that the variable name x was translated as expected.

6.7 Transition rule definitions

6.7.1 Test 8 - Passed

Test input

```
scheme test8 =
 class
   transition_system [TS]
    local
      ...
    in
      [name1] x = 0 ⟶ x′ = 1
    end
 end
```

Expected output

```
...
TRANS_REL
(x == 0 && x' == 1)
TRANS_REL_END
```

Actual output

```
...
TRANS_REL
```

```
(x == 0 && x' == 1)
TRANS_REL_END
```

6.7.2 Test 9 - Passed

Test input

```
scheme test9 =
 class
   transition_system [TS]
    local
      x : Int := 0,
      y : Bool := false
    in
      [name1] f(x) ⟶ y′ = ∼y
    end
 end
```

Expected output

```
...
TRANS_REL
(f(x) && y' == !y && x' == x)
TRANS_REL_END
```

Actual output

```
...
TRANS_REL
(f(x) && y' == !y && x' == x)
TRANS_REL_END
```

This test also shows that the function application expression $f(x)$ and the value prefix expression $\sim y$ were translated as expected.

6.7.3 Test 10 - Passed

Test input

```
scheme test10 =
 class
   transition_system [TS]
    local
       x : Int := 0
    in
       [name1] if x = 0 then true else false end ⟶ x′ = 1
       ⌷
       [name2] case x of 0 → true, _ → false end ⟶ x′ = 1
    end
 end
```

Expected output

```
...
TRANS_REL
(((x == 0 && true) || (!(x == 0) && false)) && x' == 1) ||
(((x == 0 && true) || (!(x == 0) && false)) && x' == 1)
TRANS_REL_END
```

Actual output

```
...
TRANS_REL
(((x == 0 && true) || (!(x == 0) && false)) && x' == 1) ||
(((x == 0 && true) || (!(x == 0) && false)) && x' == 1)
TRANS_REL_END
```

This test also shows that the if expression **if** $x = 0$ **then** *true* **else** *false* and the case expression **caseof** x 1 $\rightarrow$ *true*, _ $\rightarrow$ *false* **end** were translated as expected.

6.8 LTL assertion definitions

6.8.1 Test 11 - Passed

Test input

```
scheme test11 =
 class
   transition_system [TS]
    ...
   ltl_assertion
    [name1] TS ⊢ G(x = 0),
    [name1] TS ⊢ F(x = 0),
    [name1] TS ⊢ X(x = 0),
    [name1] TS ⊢ U(x = 0,x = 1),
    [name1] TS ⊢ G(F(x = 0))
 end
```

Expected output

```
...
PROP_SPEC
Globally[x == 0]
Finally[x == 0]
Next[x == 0]
[x == 0]Until[x == 1]
Globally[Finally[x == 0]]
PROP_SPEC_END
```

Actual output

```
...
PROP_SPEC
Globally[x == 0]
Finally[x == 0]
Next[x == 0]
[x == 0]Until[x == 1]
Globally[Finally[x == 0]]
```

```
PROP_SPEC_END
```

6.9 Rewriting value expressions

The following test checks whether the inference rules defining the rewriting operator ≡ from Section 4.3.5 has been implemented correctly. A manual rewriting process using the system of inference rules can be found in Appendix A.2, also using the value expression shown in the guard in the following RSL specification.

The expected output is different from the actual output in the following test. However, the two expressions are logically equivalent, so the test is deemed as passed.

6.9.1 Test 12 - Passed

Test input

```
scheme test12 =
 class
   value
    ...
   transition_system [TS]
    local
      c : Int := 0
    in
      [name] x = if y = 1 then 2 else 3 end + 4 +
                   if z = 5 then 6 else 7 end
                 ⟶ c′ = c + 1
    end
 end
```

Expected output

```
...
TRANS_REL
(((y == 1 && z == 5 && x == 2 + 4 + 6) ||
 (!(y == 1 && z == 5) &&
  ((y == 1 && !(z == 5) && x == 2 + 4 + 7) ||
   (!(y == 1 && !(z == 5)) &&
    ((!(y == 1) && z == 5 && x == 3 + 4 + 6) ||
    !(!(y == 1) && z == 5) && x == 3 + 4 + 7)))))
&& c' == c + 1)
TRANS_REL_END
...
```

Actual output

```
...
TRANS_REL
(((z == 5 && ((y == 1 && x == 2 + 4 + 6) ||
             (!(y == 1) && x == 3 + 4 + 6))) ||
 (!(z == 5) && ((y == 1 && x == 2 + 4 + 7) ||
               (!(y == 1) && x == 3 + 4 + 7))))
&& c' == c + 1)
TRANS_REL_END
...
```

Chapter 7

Demonstration of software application

In this chapter, the application of the software developed in this thesis is demonstrated by translating and model checking an RSL specification.
The chapter consists of three parts:

- The RSL specification along with an informal description of the model.
- The intermediate language translation, obtained using the developed RSL translator.
- The model checking result, obtained using the intermediate language parser and model checker.

7.1 RSL specification

For this demonstration of the application of the thesis, the model Airports.rsl has been created, representing three connected airports, *Billund*, *Frankfurt* and *Heathrow*.

Each airport has the following associated information: A plane capacity, a number of planes currently in the airport and the local weather conditions. The weather conditions may change at any time.

Planes can fly from one airport to another, but only on a number of conditions. The first airport must have a plane available for take off, and the second airport must not be at full capacity.
Furthermore, the local weather in both airports may not be stormy.

There are two properties being verified. The first is that no airport goes above its plane capacity. The second is that the number of planes in the system remains the same.
Both properties are global properties, and have therefore been negated, so they can be proven by contradiction.

7.1.1 Airports.rsl

```
scheme Airports =
class
   type
    Weather == Sunny | Stormy,
    nat = {| n : Int • n ≥ 0 |}

   value
    billund_capacity : nat = 20,
    frankfurt_capacity : nat = 30,
    heathrow_capacity : nat = 40,

    hasFreeCapacity : nat × nat × Weather → Bool
    hasFreeCapacity(p,c,w) ≡
       if w = Stormy then false else p < c end,
    hasAvailablePlane : nat × Weather → Bool
    hasAvailablePlane(p,w) ≡
       if w = Stormy then false else p > 0 end

   transition_system [TS]
    local
       billund_planes : nat := 15,
       frankfurt_planes : nat := 20,
       heathrow_planes : nat := 30,

       billund_weather : Weather := Sunny,
    frankfurt_weather : Weather := Sunny,
    heathrow_weather : Weather := Stormy

  in
    [ billund_frankfurt ]
      hasFreeCapacity(frankfurt_planes,frankfurt_capacity,
        frankfurt_weather) ∧
      hasAvailablePlane(billund_planes,billund_weather) ⟶
      frankfurt_planes′ = frankfurt_planes + 1,
      billund_planes′ = billund_planes − 1
    []
    [ frankfurt_billund ]
      hasFreeCapacity(billund_planes,billund_capacity,billund_weather) ∧
      hasAvailablePlane(frankfurt_planes,frankfurt_weather) ⟶
      billund_planes′ = billund_planes + 1,
      frankfurt_planes′ = frankfurt_planes − 1
    []
    [ billund_heathrow ]
      hasFreeCapacity(heathrow_planes,heathrow_capacity,
        heathrow_weather) ∧
      hasAvailablePlane(billund_planes,billund_weather) ⟶
      heathrow_planes′ = heathrow_planes + 1,
      billund_planes′ = billund_planes − 1
    []
    [ heathrow_billund ]
      hasFreeCapacity(billund_planes,billund_capacity,billund_weather) ∧
      hasAvailablePlane(heathrow_planes,heathrow_weather) ⟶
      billund_planes′ = billund_planes + 1,
      heathrow_planes′ = heathrow_planes − 1
    []
    [ heathrow_frankfurt ]
      hasFreeCapacity(frankfurt_planes,frankfurt_capacity,
        frankfurt_weather) ∧
      hasAvailablePlane(heathrow_planes,heathrow_weather) ⟶
      frankfurt_planes′ = frankfurt_planes + 1,
      heathrow_planes′ = heathrow_planes − 1
    []
    [ frankfurt_heathrow ]
      hasFreeCapacity(heathrow_planes,heathrow_capacity,
        heathrow_weather) ∧
      hasAvailablePlane(frankfurt_planes,frankfurt_weather) ⟶
      frankfurt_planes′ = frankfurt_planes + 1,
      frankfurt_planes′ = frankfurt_planes − 1
    []
```

```
      [ billund_SunnyWeather ]
         billund_weather ≠ Sunny ⟶ billund_weather′ = Sunny
      ▯
      [ billund_StormyWeather ]
         billund_weather ≠ Stormy ⟶ billund_weather′ = Stormy
      ▯
      [ frankfurt_SunnyWeather ]
         frankfurt_weather ≠ Sunny ⟶ frankfurt_weather′ = Sunny
      ▯
      [ frankfurt_StormyWeather ]
         frankfurt_weather ≠ Stormy ⟶ frankfurt_weather′ = Stormy
      ▯
      [ heathrow_SunnyWeather ]
         heathrow_weather ≠ Sunny ⟶ heathrow_weather′ = Sunny
      ▯
      [ billund_StormyWeather ]
         heathrow_weather ≠ Stormy ⟶ heathrow_weather′ = Stormy
   end

  ltl_assertion
   [ CapacityConstraint ]
      TS ⊢ F((billund_capacity < billund_planes) ∨
                 (frankfurt_capacity < frankfurt_planes) ∨
                 (heathrow_capacity < heathrow_planes)),
   [ PlaneConsistency ]
      TS ⊢ F((billund_planes + frankfurt_planes + heathrow_planes) ≠ 65)
end
```

7.2 Intermediate language translation

Using the developed RSL translator, the following intermediate language code is produced:

7.2.1 Airports.rtt

```
SYM_TABLE_DECL
Weather == Sunny | Stormy
nat == int n where n >= 0
```

```
const nat billund_capacity == 20
const nat frankfurt_capacity == 30
const nat heathrow_capacity == 40
bool hasFreeCapacity (nat p, nat c, Weather w)
  {return ((w == Stormy && false) || (!(w == Stormy) && p < c))}
bool hasAvailablePlane (nat p, Weather w)
  {return ((w == Stormy && false) || (!(w == Stormy) && p > 0))}
nat billund_planes
nat frankfurt_planes
nat heathrow_planes
Weather billund_weather
Weather frankfurt_weather
Weather heathrow_weather
SYM_TABLE_DECL_END

INIT_VAL
billund_planes == 15
frankfurt_planes == 20
heathrow_planes == 30
billund_weather == Sunny
frankfurt_weather == Sunny
heathrow_weather == Stormy
INIT_VAL_END

TRANS_REL
(hasFreeCapacity(frankfurt_planes,frankfurt_capacity,
  frankfurt_weather) && hasAvailablePlane(billund_planes,
  billund_weather) && frankfurt_planes' == frankfurt_planes + 1
  && billund_planes' == billund_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(hasFreeCapacity(billund_planes,billund_capacity,
  billund_weather) && hasAvailablePlane(frankfurt_planes,
  frankfurt_weather) && billund_planes' == billund_planes + 1
  && frankfurt_planes' == frankfurt_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(hasFreeCapacity(heathrow_planes,heathrow_capacity,
  heathrow_weather) && hasAvailablePlane(billund_planes,
  billund_weather) && heathrow_planes' == heathrow_planes + 1
  && billund_planes' == billund_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(hasFreeCapacity(billund_planes,billund_capacity,
  billund_weather) && hasAvailablePlane(heathrow_planes,
  heathrow_weather) && billund_planes' == billund_planes + 1
  && heathrow_planes' == heathrow_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(hasFreeCapacity(frankfurt_planes,frankfurt_capacity,
  frankfurt_weather) && hasAvailablePlane(heathrow_planes,
  heathrow_weather) && frankfurt_planes' == frankfurt_planes + 1
  && heathrow_planes' == heathrow_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(hasFreeCapacity(heathrow_planes,heathrow_capacity,
  heathrow_weather) && hasAvailablePlane(frankfurt_planes,
  frankfurt_weather) && frankfurt_planes' == frankfurt_planes + 1
  && frankfurt_planes' == frankfurt_planes - 1 &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  billund_planes' == billund_planes) ||
(billund_weather != Sunny && billund_weather' == Sunny &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(billund_weather != Stormy && billund_weather' == Stormy &&
  heathrow_weather' == heathrow_weather &&
  frankfurt_weather' == frankfurt_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(frankfurt_weather != Sunny && frankfurt_weather' == Sunny &&
  heathrow_weather' == heathrow_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(frankfurt_weather != Stormy && frankfurt_weather' == Stormy &&
  heathrow_weather' == heathrow_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(heathrow_weather != Sunny && heathrow_weather' == Sunny &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes) ||
(heathrow_weather != Stormy && heathrow_weather' == Stormy &&
  frankfurt_weather' == frankfurt_weather &&
  billund_weather' == billund_weather &&
  heathrow_planes' == heathrow_planes &&
  frankfurt_planes' == frankfurt_planes &&
  billund_planes' == billund_planes)
TRANS_REL_END

PROP_SPEC
Finally[(billund_capacity < billund_planes) ||
  (frankfurt_capacity < frankfurt_planes) ||
  (heathrow_capacity < heathrow_planes)]
Finally[(billund_planes + frankfurt_planes +
  heathrow_planes) != 65]
PROP_SPEC_END
```

7.3 Bounded model checking result

Using the intermediate language parser and model checker, the specification is checked for 10 steps and the following output is produced:

```
Attempting to solve the goal:
  Finally ([((20 < billund_planes@0) ||
     (30 < frankfurt_planes@0) ||
     (40 < heathrow_planes@0))])

[SOLVER] No solution found after 10 steps. This took 2.356s.
  Aborting search.


Attempting to solve the goal:
  Finally ([(((billund_planes@0 + frankfurt_planes@0) +
    heathrow_planes@0) != 65)])

[SOLVER] No solution found after 10 steps. This took 1.981s.
  Aborting search.
```

Based on this output, we can see that no witness has been found for either of the specified properties. Seeing as how the specified properties were negated, it is concluded that the informal properties both hold for the given bounds.

Chapter 8

User guide

This chapter contains a user guide, detailing how the software developed in this project can be installed and used.
The software has been developed for Linux systems, and has been tested on Ubuntu 14.04.3 LTS.
The source code necessary to follow this user guide can be found in the associated file *thesis.zip*.

8.1 RSL translator

This section details how to install and use the RSL translator.

8.1.1 Installation

Two things must be installed for the RSL translator to work; the rsltc tool set, which has been extended to include the translator, and Gentle, which is the compiler tool rsltc is build upon.

8.1.1.1 Gentle

Go to the *gentle-97/gentle* directory, and run the script '*build*' (i.e. type `./build` in a command console/terminal).

Go to the *gentle-97/lib* directory, and run the script '*build*'.

Go to the *gentle-97/reflex* directory, and run the script '*build*'.

8.1.1.2 rsltc

Besides Gentle, rsltc has the following dependencies:

- Cmake
- Flex
- GNU Bison
- gcc
- Automake

These can be installed with the following commands:

Cmake: `sudo apt-get install cmake`

Flex: `sudo apt-get install flex`

GNU Bison: `sudo apt-get install bison`

gcc: `sudo apt-get install gcc`

Automake: `sudo apt-get install automake`

When these are installed, go to the *rsltc* directory and run the following command, where *<GENTLE-PATH>* is the path to the *gentle-97/gentle* directory:

```
./configure -DGENTLE=<GENTLE-PATH>
```

Then go to the *rsltc/build* directory and install rsltc with the following command:

```
sudo make install
```

8.1.2 Usage

After installation, the rsltc executable will be located in the directory *rsltc/build/src*. To type check an RSL specification, use the following command:

```
./rsltc <filename>.rsl
```

To type check and translate an RSL specification to the intermediate language, use the following command:

```
./rsltc -rtt <filename>.rsl
```

This will generate the translation file `<filename>.rtt` in the same directory.

To type check an RSL specification and generate confidence conditions, use the following command:

```
./rsltc -cc <filename>.rsl
```

8.2 RT-Tester parser and model checker

This section details how to install and use the RT-Tester parser and model checker.

8.2.1 Installation

RT-Tester has a number of dependencies, which must be installed:

- `sudo apt-get install build-essential g++ python-dev`
- `sudo apt-get install autotools-dev libicu-dev libbz2-dev`
- `sudo apt-get install libboost-all-dev libclang-dev`
- `sudo apt-get install libxml2 libxml2-dev`
- `sudo apt-get install doxygen`

- `sudo apt-get install zlibc`
- `sudo apt-get install sqlite3`
- `sudo apt-get install libsqlite3-0 libsqlite3-dev`
- `sudo apt-get install libc++1 libc++-dev`

Go to the *rtttgen_ V2* directory, and use the following command:

```
./build.sh
```

If it complains about missing dependencies, simply install the dependencies in question (which should appear in the resulting output), delete the directories *build.Debug* and *build.Release*, and run `build.sh` again.

Go to the *build.Debug* directory, and use the following command:

```
make rtt-rsl
```

Building and making RT-Tester may take up to an hour to complete, depending on the speed of the computer being used.

8.2.2 Usage

After installation, the RT-Tester executable will be located in the directory *rtttgen_ V2/build.Debug/executables/rtt-rsl.*
To parse and model check a model specified in the intermediate language, use the following command:

```
./rtt-rsl <filename>.rtt <stepbound>
```

If for instance the intermediate language file *test.rtt* is located in the same directory as the executable, and the model should be checked for 10 steps, the following command is used:

```
./rtt-rsl test.rtt 10
```

The result of the model checking will the appear as output in the terminal, displaying the LTL formula being checked and the variable values in each step, in case a witness was found. An example of a result output could be:

```
Attempting to solve the goal: Finally ([success])

[SOLVER] Reached goal after 2 steps within 0.089s.

Variables in step 0:
x = 0
success = 0

Variables in step 1:
x = 1
success = 0

Variables in step 2:
x = 2
success = 1
```

Chapter 9

Further development

In this chapter, the most interesting areas for further development on this project is discussed.

9.1 Extend translatable subset of RSL

The subset of RSL which is currently accepted by the translator, is a very limiting factor in terms of which specifications can be model checked, and therefore also limits the usefulness of the developed translator. Thus the first step in continued work with the translator, could therefore be to extend the translator to include more of the available constructions in RSL.

Collections, such as the lists, sets and maps, in particular, would be a nice addition to the translator, seeing as these collections would be useful in practically all large system specifications. As pointed out in Section 4.1.5, collections with dynamic sizes would be difficult to implement in RT-Tester, but even collections with constant sizes would still be a useful addition to the translator.

Another important extension would be object arrays in RSL. By using object arrays, it would be much easier to create more involved specifications, where multiple objects are created based on some parameter.

9.2 k induction

An interesting addition to the work done is this thesis, would be to use *k induction*, as mentioned in Section 2.3.1, to perform model checking regardless of any bounds on the number of steps checked. This technique has already been used successfully with RT-Tester in the Ph.d. thesis by Linh Hong Vu [8], so his existing work could be used as a blueprint for how it can be implemented.

9.3 Reusing the RSL translator for other model checking tools

One of the ideas behind using the intermediate language, was the possibility of reusing it for other model checking tools besides RT-Tester. Though the intermediate language was designed with this in mind, it has not been tested whether it is feasible.

As such, it should always be possible to create a model based on the kind of information stored in the intermediate language. The more relevant question would rather be, how *easily* the information could be parsed into other model checking tools.

One would assume that for model checking tools which use satisfiability solving, similar to RT-Tester, it would be relatively straightforward, seeing as how the initial values and the transition relation in the intermediate language can simply be viewed as a set of constraints. However, this is certainly an area that deserves further investigation, if this project is to receive further development.

9.4 Comparing performance with other model checking tools

One of the motivating factor behind using RT-Tester for model checking, was the hope that RT-Tester could out-perform other model checking tools.
Due to the relatively small subset of RSL that is accepted by the translator, it is

difficult (or rather very tedious) to develop a model large enough, to determine whether RT-Tester can handle larger models compared to other model checking tools.

If there were more time available, or if constructions such as lists, sets and maps were to be accepted by the translator, a performance test comparing RT-Tester to SAL [18] and nuXmv [17] would be interesting.

CHAPTER 10

Conclusion

As a product of working on this thesis, a translator has been developed for a subset of RSL, which enables bounded model checking to be performed on RSL state transition systems using RT-Tester. The overall goal of this thesis has therefore been fulfilled, albeit only for a subset of RSL.

The translation has been designed in such a way, that the RSL specification is first translated to an intermediate language, which has been designed to act as a concrete syntax for RT-Tester. The intermediate language is then parsed into a model in RT-Tester, where bounded model checking is performed.

The purpose for using an intermediate language is twofold.
Firstly, it should be a readable and concise representation of a model in RT-Tester. This purpose is clearly fulfilled, as demonstrated by comparing how simple expressions are represented in the intermediate language, compared to RT-Tester (see Section 3.2.2).
Secondly, it should be possible to reuse this language (and therefore also the RSL translator) in other projects similar to this one, where other model checking tools are used. One thing that speaks in favour of this, is the fact that high level constructions (such as if/case expressions) is represented by purely logical expressions. This means that such constructions can be fed to a model checking tool exactly as for any other logical expression, regardless of whether the model checking tool supports the original construction. This is also the case

for transition relations, which are represented as one single logical expression, which acts as a constraint for any possible transition. Because of this, it could be argued that any model checking tool that uses satisfiability solvers and constraints to perform model checking (like RT-Tester), should be able to reuse the intermediate language on a conceptual level.
However, the exact syntax of the language has been designed specifically for RT-Tester. Therefore the language is not as generally applicable as possible, and it would most likely not be as straightforward to parse into other model checking tools, compared to RT-Tester.

The implementation of the translator has been tested, to show that it functions as intended in the design of the system. The application of the developed software has also been demonstrated, by performing bounded model checking on an RSL specification, using both the RSL translator and the intermediate language parser.

Finally, some of the possible further developments that could be done on this project have been discussed. In particular extending the subset of RSL which can be translated, and also using *k induction* to perform global model checking rather than bounded model checking.

Appendix A

Appendix

A.1 Example translations from RSL to the intermediate language

The following examples show translations from RSL to the intermediate language, starting with a very simple RSL specification, using only a few constructions, and gradually adding more of the translatable constructions.

A.1.1 Airport1 - Transition system and LTL assertion

Airport1.rsl

```
scheme Airport1 =
class
   transition_system [TS]
    local
       numberOfPlanes : Int := 100,
       planeCapacity : Int := 150
    in
```

```
      [ planeArrival ]
      numberOfPlanes < planeCapacity ⟶
            numberOfPlanes′ = numberOfPlanes + 1
      ⌷
      [ planeDeparture ]
      numberOfPlanes > 0 ⟶ numberOfPlanes′ = numberOfPlanes − 1
    end
   ltl_assertion
    [ CapacityConstraint ] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
    numberOfPlanes ≥ 0)
end
```

Airport1.rtt

```
SYM_TABLE_DECL
int numberOfPlanes
int planeCapacity
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
planeCapacity == 150
INIT_VAL_END

TRANS_REL
(numberOfPlanes < planeCapacity &&
  numberOfPlanes' == numberOfPlanes + 1 &&
  planeCapacity' == planeCapacity) ||
(numberOfPlanes > 0 &&
  numberOfPlanes' == numberOfPlanes - 1 &&
  planeCapacity' == planeCapacity)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.1.2 Airport2 - Explicit value definition

Airport2.rsl

```
scheme Airport2 =
class
   value
    planeCapacity : Int = 150
   transition_system [TS]
    local
      numberOfPlanes : Int := 100
    in
      [planeArrival]
      numberOfPlanes < planeCapacity ⟶
            numberOfPlanes′ = numberOfPlanes + 1
      []
      [planeDeparture]
      numberOfPlanes > 0 ⟶ numberOfPlanes′ = numberOfPlanes − 1
    end
   ltl_assertion
    [CapacityConstraint] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
    numberOfPlanes ≥ 0)
end
```

Airport2.rtt

```
SYM_TABLE_DECL
const int planeCapacity == 150
int numberOfPlanes
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
INIT_VAL_END

TRANS_REL
(numberOfPlanes < planeCapacity &&
  numberOfPlanes' == numberOfPlanes + 1) ||
(numberOfPlanes > 0 && numberOfPlanes' == numberOfPlanes - 1)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.1.3 Airport3 - Subtype definition

Airport3.rsl

```
scheme Airport3 =
class
   type
    nat = {| n : Int • n ≥ 0 |}
   value
    planeCapacity : nat = 150
   transition_system [TS]
    local
      numberOfPlanes : nat := 100
    in
      [planeArrival]
      numberOfPlanes < planeCapacity ⟶
            numberOfPlanes′ = numberOfPlanes + 1
      ⌷
      [planeDeparture]
      numberOfPlanes > 0 ⟶ numberOfPlanes′ = numberOfPlanes − 1
    end
   ltl_assertion
    [CapacityConstraint] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
    numberOfPlanes ≥ 0)
end
```

Airport3.rtt

```
SYM_TABLE_DECL
nat == int n where n >= 0
const nat planeCapacity == 150
nat numberOfPlanes
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
INIT_VAL_END

TRANS_REL
(numberOfPlanes < planeCapacity &&
  numberOfPlanes' == numberOfPlanes + 1) ||
(numberOfPlanes > 0 && numberOfPlanes' == numberOfPlanes - 1)
```

```
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.1.4 Airport4 - Explicit function definition and function application expression

Airport4.rsl

```
scheme Airport4 =
class
   type
    nat = {| n : Int • n ≥ 0 |}
   value
    planeCapacity : nat = 150,
    hasFreeCapacity : nat × nat → Bool
    hasFreeCapacity(p,c) ≡ p < c,
    hasAvailablePlane : nat → Bool
    hasAvailablePlane(p) ≡ p > 0
   transition_system [TS]
    local
      numberOfPlanes : nat := 100
    in
      [planeArrival]
      hasFreeCapacity(numberOfPlanes,planeCapacity) ⟶ numberOfPlanes′ =
      numberOfPlanes + 1
      ▯
      [planeDeparture]
      hasAvailablePlane(numberOfPlanes) ⟶
            numberOfPlanes′ = numberOfPlanes − 1
    end
   ltl_assertion
    [CapacityConstraint] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
    numberOfPlanes ≥ 0)
end
```

Airport4.rtt

```
SYM_TABLE_DECL
nat == int n where n >= 0
const nat planeCapacity == 150
bool hasFreeCapacity (nat c,nat p) {return p < c}
bool hasAvailablePlane (nat p) {return p > 0}
nat numberOfPlanes
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
INIT_VAL_END

TRANS_REL
(hasFreeCapacity(numberOfPlanes,planeCapacity) &&
  numberOfPlanes' == numberOfPlanes + 1) ||
(hasAvailablePlane(numberOfPlanes) &&
  numberOfPlanes' == numberOfPlanes - 1)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.1.5 Airport5 - If expression and case expression

Airport5.rsl

```
scheme Airport5 =
class
   type
    nat = {| n : Int • n ≥ 0 |}
   value
    planeCapacity : nat = 150,
    hasFreeCapacity : nat × nat × Bool → Bool
    hasFreeCapacity(p,c,b) ≡
       if b then false else p < c end,
    hasAvailablePlane : nat × Bool → Bool
    hasAvailablePlane(p,b) ≡
       case b of
        true → false,
        _ → p > 0
```

```
      end
   transition_system [TS]
    local
      numberOfPlanes : nat := 100,
      badWeatherConditions : Bool := false
    in
      [planeArrival]
      hasFreeCapacity(numberOfPlanes,planeCapacity,badWeatherConditions)
            ⟶ numberOfPlanes′ =
      numberOfPlanes + 1
      ▯
      [planeDeparture]
      hasAvailablePlane(numberOfPlanes,badWeatherConditions) ⟶
            numberOfPlanes′ = numberOfPlanes − 1
      ▯
      [goodWeather] badWeatherConditions ⟶ badWeatherConditions′ =
      false
      ▯
      [badWeather] ∼badWeatherConditions ⟶ badWeatherConditions′ = true
    end
   ltl_assertion
    [CapacityConstraint] TS ⊢ G(numberOfPlanes ≤ planeCapacity ∧
    numberOfPlanes ≥ 0)
end
```

Airport5.rtt

```
SYM_TABLE_DECL
nat == int n where n >= 0
const nat planeCapacity == 150
bool hasFreeCapacity (nat p,nat c,bool b)
  {return ((b && false) || (!(b) && p < c))}
bool hasAvailablePlane (nat p,bool b)
  {return (b && false) || (!(b) && p > 0)}
nat numberOfPlanes
bool badWeatherConditions
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
badWeatherConditions == false
INIT_VAL_END
```

```
TRANS_REL
(hasFreeCapacity(numberOfPlanes,planeCapacity,
  badWeatherConditions) &&
  numberOfPlanes' == numberOfPlanes + 1 &&
   badWeatherConditions' == badWeatherConditions) ||
(hasAvailablePlane(numberOfPlanes,badWeatherConditions) &&
  numberOfPlanes' == numberOfPlanes - 1 &&
  badWeatherConditions' == badWeatherConditions) ||
(badWeatherConditions && badWeatherConditions' == false &&
  numberOfPlanes' == numberOfPlanes) ||
(!badWeatherConditions && badWeatherConditions' == true &&
  numberOfPlanes' == numberOfPlanes)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.1.6 Airport6 - Variant type definition

Airport6.rsl

```
scheme Airport6 =
class
  type
   Weather == Sunny | Cloudy | Stormy | Hurricane,
   nat = {| n : Int • n ≥ 0 |}
  value
   planeCapacity : nat = 150,
   hasFreeCapacity : nat × nat × Weather → Bool
   hasFreeCapacity(p,c,w) ≡
     if w = Stormy ∨ w = Hurricane then false else p < c end,
   hasAvailablePlane : nat × Weather → Bool
   hasAvailablePlane(p,w) ≡
     case w of
      Stormy → false,
      Hurricane → false,
      _ → p > 0
     end
  transition_system [TS]
   local
```

numberOfPlanes : nat := 100,
weatherConditions : Weather := Sunny
in
[planeArrival]
hasFreeCapacity(numberOfPlanes,planeCapacity,weatherConditions) $\longrightarrow$ numberOfPlanes$'$ =
numberOfPlanes + 1
$\Box$
[planeDeparture]
hasAvailablePlane(numberOfPlanes,weatherConditions) $\longrightarrow$
numberOfPlanes$'$ = numberOfPlanes $-$ 1
$\Box$
[SunnyWeather] weatherConditions $\neq$ Sunny $\longrightarrow$ weatherConditions$'$ =
Sunny
$\Box$
[CloudyWeather] weatherConditions $\neq$ Cloudy $\longrightarrow$ weatherConditions$'$ =
Cloudy
$\Box$
[StormyWeather] weatherConditions $\neq$ Stormy $\longrightarrow$ weatherConditions$'$ =
Stormy
$\Box$
[HurricaneWeather] weatherConditions $\neq$ Hurricane $\longrightarrow$
weatherConditions$'$ =
Hurricane
end
ltl_assertion
[CapacityConstraint] TS $\vdash$ G(numberOfPlanes $\leq$ planeCapacity $\wedge$
numberOfPlanes $\geq$ 0)
end

Airport6.rtt

```
SYM_TABLE_DECL
Weather == Sunny | Cloudy | Stormy | Hurricane
nat == int n where n >= 0
const nat planeCapacity == 150
bool hasFreeCapacity (nat p,nat c,Weather w)
  {return ((w == Stormy || w == Hurricane && false) ||
    (!(w == Stormy || w == Hurricane) && p < c))}
bool hasAvailablePlane (nat p,Weather w)
  {return (w == Stormy && false) || (w == Hurricane && false) ||
    (!(w == Hurricane) && !(w == Stormy) && p > 0)}
nat numberOfPlanes
```

```
Weather weatherConditions
SYM_TABLE_DECL_END

INIT_VAL
numberOfPlanes == 100
weatherConditions == Sunny
INIT_VAL_END

TRANS_REL
(hasFreeCapacity(numberOfPlanes,planeCapacity,weatherConditions)
  && numberOfPlanes' == numberOfPlanes + 1 &&
  weatherConditions' == weatherConditions) ||
(hasAvailablePlane(numberOfPlanes,weatherConditions) &&
  numberOfPlanes' == numberOfPlanes - 1 &&
  weatherConditions' == weatherConditions) ||
(weatherConditions != Sunny && weatherConditions' == Sunny &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Cloudy && weatherConditions' == Cloudy &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Stormy && weatherConditions' == Stormy &&
  numberOfPlanes' == numberOfPlanes) ||
(weatherConditions != Hurricane &&
  weatherConditions' == Hurricane &&
  numberOfPlanes' == numberOfPlanes)
TRANS_REL_END

PROP_SPEC
Globally[numberOfPlanes <= planeCapacity && numberOfPlanes >= 0]
PROP_SPEC_END
```

A.2 Example of rewriting value expressions containing if expressions

The following is an example of how a value expression containing if expressions can be rewritten and translated using the inference rules defined in Section 4.3.5. The approach used here, is to find rule that matches a given expression, and then filling in the parts of the rule which is known, and assigning variables to the unknown parts. Also note that many trivial steps have been skipped, where expressions already are in normal form, in order to keep this example a bit shorter.

Consider the following value expression:

$$\text{x} = \textbf{if } \text{y} = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} + 4 + \textbf{if } \text{z} = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end}$$

The expression matches the rule (4.51):

$$\frac{\begin{array}{l} x \equiv A1,\ \textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} + 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \equiv A2, \\ A1 = A2 \equiv A3, \\ A3 \rhd A4 \end{array}}{x = \textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} + 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \rhd\ A4}$$

The expression $x \equiv A1$ matches the rule (4.65):

$$\frac{}{x \equiv x}$$

and so $A1$ is x.

The expression $\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} + 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \equiv A2$ matches the rule (4.65):

$$\frac{\begin{array}{l} \textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} \equiv B1, \\ 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \equiv B2,\ B1 + B2 \equiv A2 \end{array}}{\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} + 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \equiv A2}$$

The expression $\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} \equiv B1$ matches the rule (4.65):

$$\frac{}{\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} \equiv \textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end}}$$

and so $B1$ is $\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end}$

The expression $4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \equiv B2$ matches the rule (4.61):

$$\frac{\begin{array}{l} z = 5 \equiv C1,\ 6 \equiv C2,\ 7 \equiv C3, \\ 4 + C2 \equiv C4,\ 4 + C3 \equiv C5 \end{array}}{\begin{array}{l} 4 + \textbf{if } z = 5 \textbf{ then } 6 \textbf{ else } 7 \textbf{ end} \\ \equiv \\ \textbf{if } C1 \textbf{ then } C4 \textbf{ else } C5 \textbf{ end} \end{array}}$$

The expression $z = 5 \equiv C1$ matches the rule (4.64):

$$\frac{}{z = 5 \equiv z = 5}$$

and so $C1$ is $z = 5$.

The expressions $6 \equiv C2$ and $7 \equiv C3$ both match the rule (4.65):

$$\frac{}{6 \equiv 6}$$

$$\frac{}{7 \equiv 7}$$

and so $C2$ is 6 and $C3$ is 7.

The expressions $4 + C2 \equiv C4$ and $4 + C3 \equiv C5$ both match the rule (4.64):

$$\frac{}{4 + 6 \equiv 4 + 6}$$

$$\frac{}{4 + 7 \equiv 4 + 7}$$

and so $C4$ is $4 + 6$ and $C4$ is $4 + 7$.

Using $C1$, $C4$ and $C5$, we find that the value of $B2$ is **if** $z = 5$ **then** $4 + 6$ **else** $4 + 7$ **end**

Using $B1$ and $B2$, we get the expression $B1 + B2 \equiv A2$ which matches the rule (4.60):

$$\frac{\begin{array}{c}2 \equiv D1,\ 3 \equiv D2,\ 4+6 \equiv D3,\ 4+7 \equiv D4,\ y = 1 \equiv D5,\ z = 5 \equiv D6 \\ D1 + D3 \equiv D7,\ D1 + D4 \equiv D8,\ D3 + D2 \equiv D9,\ D2 + D4 \equiv D10\end{array}}{\begin{array}{l}\textbf{if } y = 1 \textbf{ then } 2 \textbf{ else } 3 \textbf{ end} \\ + \\ \textbf{if } z = 5 \textbf{ then } 4 + 6 \textbf{ else } 4 + 7 \textbf{ end} \\ \ \equiv \\ \textbf{if } D5 \wedge D6 \textbf{ then } D7 \textbf{ else} \\ \quad \textbf{if } D5 \wedge \sim D6 \textbf{ then } D8 \textbf{ else} \\ \qquad \textbf{if } \sim D5 \wedge D6 \textbf{ then } D9 \textbf{ else } D10 \\ \qquad \textbf{end} \\ \quad \textbf{end} \\ \textbf{end}\end{array}}$$

To simplify this process a bit, lets say we already found that
$D1$ is 2

$D2$ is 3
$D3$ is $4+6$
$D4$ is $4+7$
$D5$ is $y=1$
$D6$ is $z=5$
$D7$ is $1+4+6$
$D8$ is $1+4+7$
$D9$ is $4+6+3$
$D10$ is $3+4+7$
This can be found in a similar way to $C1$, $C2$, $C3$, $C4$ and $C5$.

Using this, we get that $A2$ is

```
**if** $y = 1 \wedge z = 5$ **then** $2+4+6$ **else**
  **if** $y = 1 \wedge \sim (z = 5)$ **then** $2+4+7$ **else**
    **if** $\sim (y = 1) \wedge z = 5$ **then** $4+6+3$ **else** $3+4+7$
    **end**
  **end**
**end**
```

In the original rule, we have $A1 = A2 \equiv A3$. Using the values of $A1$ and $A2$, we get:

```
$x =$
**if** $y = 1 \wedge z = 5$ **then** $2+4+6$ **else**
  **if** $y = 1 \wedge \sim (z = 5)$ **then** $2+4+7$ **else**
    **if** $\sim (y = 1) \wedge z = 5$ **then** $4+6+3$ **else** $3+4+7$
    **end**
  **end**
**end**
$\equiv$
$A3$
```

This expression matches the rule (4.61):

```
$y = 1 \wedge z = 5 \equiv E1,$
$2 + 4 + 6 \equiv E2,$
**if** $y = 1 \wedge \sim (z = 5)$ **then** $2 + 4 + 7$ **else**
  **if** $\sim (y = 1) \wedge z = 5$ **then** $4 + 6 + 3$ **else** $3 + 4 + 7$
  **end**
**end**
 $\equiv E3$
$x = E2 \equiv E4$
$x = E3 \equiv E5$
--------------------------------------------------
$x =$
**if** $y = 1 \wedge z = 5$ **then** $2 + 4 + 6$ **else**
  **if** $y = 1 \wedge \sim (z = 5)$ **then** $2 + 4 + 7$ **else**
    **if** $\sim (y = 1) \wedge z = 5$ **then** $4 + 6 + 3$ **else** $3 + 4 + 7$
    **end**
  **end**
**end**
 $\equiv$
**if** $E1$ **then** $E4$ **else** $E5$ **end**
```

Again to simplify this process a bit, lets say we already found that
$E1$ is $y = 1 \wedge z = 5$
$E2$ is $2 + 4 + 6$
$E3$ is

```
**if** $y = 1 \wedge \sim (z = 5)$ **then** $2 + 4 + 7$ **else**
  **if** $\sim (y = 1) \wedge z = 5$ **then** $4 + 6 + 3$ **else** $3 + 4 + 7$
  **end**
**end**
```

$E4$ is $x = 2 + 4 + 6$
$E5$ is

```
**if** $y = 1 \wedge \sim (z = 5)$ **then** $x = 2 + 4 + 7$ **else**
  **if** $\sim (y = 1) \wedge z = 5$ **then** $x = 4 + 6 + 3$ **else** $x = 3 + 4 + 7$
  **end**
**end**
```

Using this, we get that $A3$ is

```
**if** $y = 1 \wedge z = 5$ **then** $x = 2 + 4 + 6$ **else**
  **if** $y = 1 \wedge \sim (z = 5)$ **then** $x = 2 + 4 + 7$ **else**
    **if** $\sim (y = 1) \wedge z = 5$ **then** $x = 4 + 6 + 3$ **else** $x = 3 + 4 + 7$
    **end**
  **end**
**end**
```

Going back to the first rule we used, we now have the translation expression

```
**if** $y = 1 \wedge z = 5$ **then** $x = 2 + 4 + 6$ **else**
  **if** $y = 1 \wedge \sim (z = 5)$ **then** $x = 2 + 4 + 7$ **else**
    **if** $\sim (y = 1) \wedge z = 5$ **then** $x = 4 + 6 + 3$ **else** $x = 3 + 4 + 7$
    **end**
  **end**
**end**
$\rhd$
$A4$
```

This matches the rule (4.55):

$$\frac{\begin{array}{l} y = 1 \wedge z = 5 \rhd F1,\ x = 2 + 4 + 6 \rhd F2, \\ \textbf{if}\ y = 1 \wedge \sim (z = 5)\ \textbf{then}\ x = 2 + 4 + 7\ \textbf{else} \\ \quad \textbf{if}\ \sim (y = 1) \wedge z = 5\ \textbf{then}\ x = 4 + 6 + 3\ \textbf{else}\ x = 3 + 4 + 7 \\ \quad \textbf{end} \\ \textbf{end} \\ \ \rhd F3 \end{array}}{\begin{array}{l} \textbf{if}\ y = 1 \wedge z = 5\ \textbf{then}\ x = 2 + 4 + 6\ \textbf{else} \\ \quad \textbf{if}\ y = 1 \wedge \sim (z = 5)\ \textbf{then}\ x = 2 + 4 + 7\ \textbf{else} \\ \qquad \textbf{if}\ \sim (y = 1) \wedge z = 5\ \textbf{then}\ x = 4 + 6 + 3\ \textbf{else}\ x = 3 + 4 + 7 \\ \qquad \textbf{end} \\ \quad \textbf{end} \\ \textbf{end} \\ \rhd \\ ((F1\ \&\&\ F2)\ ||\ (!(F1)\ \&\&\ F3)) \end{array}}$$

Again to simplify this process a bit, lets say we already found that
$F1$ is $y == 1\ \&\&\ z == 5$

$F2$ is $x == 2 + 4 + 6$
$F3$ is
$((y == 1 \,\&\&\, !(z == 5) \,\&\&\, x == 2 + 4 + 7) \,||$
$\quad (!(y == 1 \,\&\&\, !(z == 5)) \,\&\&\, ((!(y == 1) \,\&\&\, z == 5 \,\&\&\, x == 3 + 4 + 6) \,||$
$\quad !(!(y == 1) \,\&\&\, z == 5) \,\&\&\, x == 3 + 4 + 7)))$

Using $F1$, $F2$ and $F3$ in the expression $((F1 \,\&\&\, F2) \,||\, (!(F1) \,\&\&\, F3))$, we now know that the final result $A4$ is
$((y == 1 \,\&\&\, z == 5 \,\&\&\, x == 2 + 4 + 6) \,||$
$(!(y == 1 \,\&\&\, z == 5) \,\&\&$
$\quad ((y == 1 \,\&\&\, !(z == 5) \,\&\&\, x == 2 + 4 + 7) \,||$
$\quad\quad (!(y == 1 \,\&\&\, !(z == 5)) \,\&\&$
$\quad\quad\quad ((!(y == 1) \,\&\&\, z == 5 \,\&\&\, x == 3 + 4 + 6) \,||$
$\quad\quad\quad !(!(y == 1) \,\&\&\, z == 5) \,\&\&\, x == 3 + 4 + 7)))))$

thus concluding this process of rewriting and translating.

Bibliography

[1] The RAISE Language Group: Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, Kim Ritter Wagner, *The RAISE Specification Language*, ISBN: 0137528337, Prentice Hall International (UK) Ltd, Campus 400, Maylands Avenue, UK, 1992.

[2] Chris George, *RAISE Tool User Guide*, April 17, 2008 UNU-IIST Report No. 227.

[3] Juan Ignacio Perna and Chris George, *Model checking RAISE specifications*, November, 2006 UNU-IIST Report No. 331.

[4] Friedrich Wilhelm Schröer, *The GENTLE Compiler Construction System*, Metarga, Berlin, 2005.

[5] Anne E. Haxthausen and Jan Peleska, *Model Checking and Model-based Testing in the Railway Domain*, R. Drechsler, U Kühne (eds.), Formal Modelling and Verification of Cyber-Physical Systems, DOI 10.1007/978-3-658-09994-7_4, Springer Fachmedien Wiesbaden 2015.

[6] Verified Systems International GmbH, *RT-Tester Model-Based Test Case and Test Data Generator*, Version 9.0-1.3.0, User Manual, Document-Id: Verified-INT-003-2012.

[7] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, 2008, ISBN 978-0-262-02649-9, The MIT Press, Cambridge, Massachusetts, London, England.

[8] Linh Hong Vu, *Formal Development and Verification of Railway Control Systems*, PHD-2015-395, DTU Compute. 2015,

[9] Anne E. Haxthausen, *An Introduction to Formal Methods for the Development of Safety-critical Applications*, DTU Informatics, August 30, 2010.

[10] Anne E. Haxthausen, *Lecture notes on The RAISE Development Method*, April 1999.

[11] Anne E. Haxthausen, *Introduction to RAISE*, Lecture slides, DTU Informatics, 2015.

[12] ANTLR, http://www.antlr.org/about.html, last checked: 12-01-2016.

[13] Kim Sørensen, *Model Checking RAISE Specifications using nuXmv*, DTU Compute, 2015.

[14] nuXmv, https://nuxmv.fbk.eu/, last checked: 12-01-2016.

[15] Lex and Yacc, http://dinosaur.compilertools.net/, last checked: 14-01-2016.

[16] S. Owre, S. Rajan, J. M. Rushby, N Shankar and M. Srivas, *PVS: Combining Specification, Proof Checking and Model Checking*, Computer Science Laboratory, SRI International, 1996.

[17] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, *nuXmv 1.0 User Manual*, FBK - Via Sommarive 18, 38055 Povo (Trento) - Italy, 2014.

[18] Leonardo de Moura, Sam Owre and N. Shankar. *The SAL Language Manual*, CSL Technical Report SRI-CSL-01-02 (Rev. 2), August, 2003.