MASTER

The formal verification of Rose RealTime software models using Petri nets

van Leeuwen, M.

Award date:
2007

Link to publication
The formal verification of Rose RealTime software models using Petri nets

By
Marcel van Leeuwen

A thesis submitted to the Eindhoven University of Technology in conformity with the requirements for the degree of
Master of Science

Supervisors:
Prof.dr. K.M. van Hee (TU/e)
dr. L. Somers (Océ-Technologies B.V.)

Venlo, June 2007
Abstract

The research presented in this Master Thesis aims at defining formal means by which Rose RealTime (UML) software models can be verified for correctness. To do so a translation of UML software models into classic Petri nets is presented. Thirteen patterns are formulated to define the translation. Two of these concern the translation of UML structure diagrams, seven the translation of UML state machines, and the remaining four concern the translation of a strictly structural version of the C++ code defined in state machines. With structural code analysis we mean that the code has been analyzed for the presence of iteration, selection and some specific Rose RealTime statements, but that variable values are not taken into account.

The size of the Petri nets generated from real-life Rose RealTime software models, and the state space explosion problem that resulted from it, obliged us to define means by which the size of the Petri nets can be reduced to a level at which formal verification becomes feasible. These means are defined as ten optimization and five structural reduction rules.

Together with developers that use Rose RealTime for the implementation of their software systems a number of analysis tasks have been formulated. These are implemented as Petri net analysis tasks with the analysis tool LoLa. To obtain useful analysis results the tasks are chosen in such a way that a counterexample trace is generated when verification of a certain property fails.

The creation of a counterexample Petri net trace does not help a developer in tracing an error in the software system. Therefore, a translation of these traces into UML message sequence charts is defined. The message sequence chart clearly visualizes the messages that are exchanged between the different components in the system to Rose RealTime software developers.

The process of translating and analyzing Rose RealTime software models is implemented in a toolkit. The toolkit is used to analyze a small-scale Rose RealTime model and to analyze three real-life large-scale software models provided by Océ-Technologies B.V.
Acknowledgements

The study Business Information Systems at the Eindhoven University of Technology is concluded with a master project. The Master Thesis at hand is one of the most tangible results of this project. Before actually presenting the results of the eight month project I would like to thank those who in someway participated in its realization.

I would like to thank Prof.dr. Kees van Hee and dr. Lou Somers for their ideas, hints, comments and support during my research. In the first stage of my project I had some difficulties mastering the Rose RealTime software tool and all its peculiarities, I want to thank Ton Janssen and Ben van Rens for helping me understand the details.

The libraries on which some parts of the implemented toolkit rely are mainly developed and maintained by drs. Reinier Post, whom I want to thank for his support on how to use the libraries and for helping me as much as possible when a problem, mostly due to a misinterpretation from my side, arose. To Jan-Martijn van der Werf and dr. Marc Voorhoeve I express my gratitude for their help with the definition of Petri net reduction rules and to Lusine Hakobyan for her detailed and thorough reviews of this document.

Finally, I take this opportunity to deeply thank my parents and brother for their continuous support during the past six years.

Marcel van Leeuwen
Venlo, June 2007
# Contents

Abstract i  

Acknowledgements iii  

1 Introduction 1  

1.1 Problem Description ............................................. 1  
1.2 Goals ........................................................................ 2  
1.3 Outline ....................................................................... 3  

2 Related Work 5  

3 Rose RealTime 7  

3.1 Background ............................................................. 7  
3.2 Introduction ............................................................ 8  
3.3 Logical View ........................................................... 9  

3.3.1 Introduction .......................................................... 9  
3.3.2 Protocol ............................................................... 11  
3.3.3 Class .................................................................... 12  
3.3.4 Capsule ............................................................... 12  
3.3.5 Structure diagram .................................................. 13  
3.3.6 State diagram ....................................................... 19  
3.4 Event handling ......................................................... 25  

4 Petri net preliminaries 27  

4.1 Petri net introduction ................................................. 27  
4.2 Petri net tools .......................................................... 28  
4.3 Analysis methods ...................................................... 29  

5 Rose RealTime model translation 33  

5.1 Usage guidelines ...................................................... 33  
5.2 Model translation ...................................................... 34  

5.2.1 Structural translation .............................................. 34  
5.2.2 Behavioral translation ............................................. 38  
5.2.3 Code translation .................................................... 48  
5.2.4 Transition atomicity .............................................. 58  
5.3 Limitations ............................................................... 58  

5.3.1 Static model analysis .............................................. 59  
5.3.2 Thread analysis ..................................................... 59  
5.3.3 Message queue ..................................................... 59  

5.4 Translation of Petri net analysis results back into RoseRT 60  

6 Optimizations & Reductions 63  

6.1 Optimizations ............................................................ 63  

6.1.1 H01: Disregard unconnected ports ................................. 63  
6.1.2 H02: Disregard unconnected capsule roles ....................... 64  
6.1.3 H03: Relay ports disregarded .................................... 65  
6.1.4 H04: Translation of in signals disregarded ....................... 65  
6.1.5 H05: Limited capsule role cardinality ............................ 66
Introduction

This thesis describes a method to formally verify a certain class of software models. This class of software models is designed with the Rose RealTime tool, in which software systems are modeled with the Unified Modeling Language (UML).

Due to the lack of a clear and formal description of the UML semantics there are currently no means by which a formal proof of correctness of an UML model can be given. Translation of an UML model into a mathematically well-defined paradigm for which formal analysis methods do exist, allows for the definition of analysis tasks with which the correctness of a model can be proven. This thesis describes such a translation of Rose RealTime models into classic Petri nets.

The translation of Rose RealTime UML models into Petri nets concerns the translation of specific UML constructs that define the interaction between the components of a software model, and the translation of C++ code fragments in which actual system behavior is specified. The translation of C++ code is static, which means that all ‘coloring’ is removed from the code, i.e. iteration or selection statements are identified but their conditions are not translated into the Petri net.

Four analysis tasks that enable a developer to formally verify the inner workings of a model are introduced. The toolkit in which these tasks are implemented is designed to be easily extended with additional verification tasks. All tasks return a counterexample trace when verification fails, this trace is used to visualize the error to the developer. An additional translation of Petri net traces into Rose RealTime models is, therefore, defined: a Petri net trace is translated into a UML message sequence chart. This message sequence chart presents a comprehensive overview of signals being exchanged between the different system components in the order that resulted in the error state.

The translation is applied on a small scale example and on three real-life models provided by Océ-Technologies B.V. The analysis results are presented and serve as a proof-of-concept but also reveal some limitations of the defined translation.

The next sections will further introduce the problem and present the outline of this thesis.

1.1 Problem Description

Developers all around the world use Rational Rose RealTime to develop their software systems. The Rose RealTime tool enables developers to create and maintain their software at a high level of abstraction. The models are created using the modeling constructs specified in the UML 2.0 standard [31] defined by the Object Management Group (OMG). The UML dialect used in Rose RealTime is referred to as UML-RT.

UML is a language to model software systems with. Modeling a software
CHAPTER 1. INTRODUCTION

system is done using various viewpoints: from user interaction to detailed designs for implementation. One of the main reasons for the industry-wide adoption of UML as software modeling language is exactly this wide variety of viewpoints that allows developers to model a system as if it were actually in its environment.

But, as suggested by B. Selic in [44] there is a drawback to the availability of all these viewpoints: UML software models “tend to be highly diverse and much more complex than most mathematical formalisms can handle”. The unclarity of UML semantics in combination with the availability of the different viewpoints makes it that proving correctness of UML software models is a difficult and laborious task. On the other hand, with the use of flow graphs it is shown in [13] that the visual notation of UML-RT, in fact, has a deep mathematical foundation. The presence of a mathematical foundation suggests that formal verification should, at least up to a certain level, be possible.

The growing use of tools that use UML to model and create entire software systems, and the growing dependence of society on these systems, makes the need for formal verification methods more and more apparent.

1.2 Goals

The goal of the research denoted in this thesis is “to define a translation of UML-RT models into Petri nets that allow for the identification of errors in the original Rose RealTime model”.

A number of subgoals are formulated that need to be reached to achieve fulfillment of the main goal:

1. **Translation of Rose RealTime models into classic Petri nets** A formal translation of Rose RealTime, or UML-RT, models into Petri nets needs to be defined and implemented. The translation will concern both the structural and behavioral elements of a Rose RealTime model, including certain C++ code fragments.

2. **Identification of relevant Petri net analysis tasks** In consultation with Rose RealTime developers it will be decided what kind of correctness questions can typically be defined for a Rose RealTime model. These questions will be implemented as Petri net analysis tasks using well-known and well-established Petri net analysis methods that have, over the years, been developed by numerous researchers.

3. **Translation of analysis results into Rose RealTime models** If verification of a certain property, i.e. by means of an analysis task, fails on a certain Petri net this means that there might be an error in the Rose RealTime model. All analysis tasks should be implemented such that a counterexample trace is returned when verification fails. This trace is translated into a message sequence chart in the Rose RealTime model, with which a developer can easily trace the error.
Preferably will all subgoals be implemented in one toolkit that can be used by any Rose RealTime developer. As Rose RealTime developers normally do not have knowledge of Petri net theories should Petri net internals be hidden, i.e. analysis results should only be visualized in the Rose RealTime model.

Specific contributions created to achieve the goals are:

- a Petri net based analysis method that allows Rose RealTime developers to formally verify correctness of Rose RealTime models without the need to understand Petri nets. Implementation of the method required the creation of:
  - a library to create intermediary meta models from Rose RealTime models
  - a library that translates meta models into Petri nets
  - a library that parses the results generated by Petri net analysis tools and translates these results back into the original Rose RealTime model
  - a program that combines the previous components in such a way that a comprehensible user interface is presented to the user that allows for the execution of the available tasks
- the translation and analysis tasks performed on the coffee machine example as provided by the Rose RealTime tool
- the translation and analysis tasks performed on a large, real-life model provided by Océ-Technologies B.V.
- statistics on how much the different UML constructs are applied in different Rose RealTime models

1.3 Outline

The structure of the thesis as presented in this section follows the different steps of the research project.

Chapter 2 presents an overview of related research projects. Chapter 3 and Chapter 4 introduce the RoseRT tool and the basic concepts of classic Petri nets, respectively.

The definition of the translation of Rose RealTime models into Petri nets is presented in Chapter 5 followed by a presentation of a number of translation optimizations and reductions in Chapter 6. Chapter 7 presents the implementation of theories presented in the previous two chapters. The translation and analysis of a small-scale example model is presented in Chapter 8 followed by the analysis results of some large scale real-life models in Chapter 9.

Finally, Chapter 10 concludes the thesis with a presentation of the conclusions and list of recommendations.
Related Work

Other attempts to formally translate, parts of, Rose RealTime models, or UML models in general, into mathematical formalisms have been undertaken. These attempts will be presented in this chapter.

Tests to verify the boundedness of UML-RT communication buffers are presented in [22]. For the translation into Petri nets it would be of great help if boundedness could indeed be guaranteed, however, a certain level of abstraction is needed because unboundedness is undecidable as program code for instance can contain arbitrary statements and thus simulate a Turing machine. The program code is, therefore, ignored in their test for boundedness, which makes the test results not applicable to the translation as defined in this thesis as it does take program code into account.

Application of Time Petri nets (TPN), as introduced by Merlin and Farber [23], for the translation of event-driven real-time systems is performed by Z. Gu and K. G. Shin [14]. In their translation Petri nets serve as an intermediary format that is to be further translated into timed automata (TA). Much effort has been put in the translation of TPNs into semantically equivalent TA and the verification thereof. Only little effort has been put in the method on how to generate TPNs from UML models, which is why the developed theories cannot be related to our translation as too many of the UML elements are abstracted over.

In [5, 19, 20, 24, 47] translations to Generalized Stochastic Petri nets (GSPNs), another class of Timed Petri nets, are defined. The translation of a Message Sequence Chart (MSC) is presented in [5], MSCs are not taken into account in our translation.

The translation in [17] adds additional elements to RoseRT state machines, which is why many of their theories can not be applied as the translation defined in this thesis is kept as generic as possible. For our purposes does the translation defined in [24] focus too much on performance measurements which makes it unnecessary difficult. The translations defined in [19, 20] have overlaps with our translation. Application of translations to GSPNs should carefully be considered as these are mainly defined with performance evaluations being their main purpose.

Another approach is to take advantage of the object-oriented structure available in UML models by defining a translation into Object Petri nets (OPNs). Hints on how to translate interaction and state diagrams into OPNs are given in [2]. Actual translations are presented in [36] and [37]. A major disadvantage is the current lack of analysis and simulation tools that support this class of Petri nets. Availability of analysis tools is a necessary prerequisite for the achievement of the goals as defined in Section 1.2. The main theories and formalisms developed and defined in this translation could, therefore, not be applied in our translation.

Colored Petri nets are another class of Petri nets, which researchers have been using for the definition of Rose RealTime, or UML, semantics. The approach taken in [8] focuses on the translation of sequence diagrams into colored...
Petri nets. Although overlap with the translation of structure diagrams in combination with state machines is present, does the translation itself ignore some elements essential to our translation. The translation presented in [16] is performed using an approach similar to ours except that their main goal is system simulation, while we are trying to verify system behavior using mathematically defined properties. For UML activity diagrams CPN semantics are defined in [45].

Another approach to formalizing UML using colored Petri nets is undertaken in [3]. The authors introduce an approach very different from the others: the Customization Rules (CR) approach. It is worthwhile to shortly explain their approach: they argue that one universal translation that can be applied to all Rose RealTime models is not feasible. They reason that users should define a set of rules, hence customization, in which they specify the semantics as they interpreted them for the different modeling elements. Only after the specification of these custom semantics is it possible to make a translation.

In [10] semantics are defined using classic Petri nets. The UML, and thus Rose RealTime, elements and diagrams that are taken into account in their translation partly overlap with the elements and diagrams relevant to our translation. Some of the defined translation constructs could, therefore, be applied in our translation.

Until here only Petri net-based translations have been discussed, but researchers have also investigated the possibilities of formalizing Rose RealTime or UML models using other, non Petri net based, mathematical formalisms. These approaches will be shortly introduced here.

Defining semantics of UML state machines with the use of structured graph transformations, like Petri nets a thoroughly studied area, is the approach undertaken in [21]. State diagrams are represented as graphs, the transitions between elements with transformation rules. In [12] an attempt to capture UML-RT semantics with flow-graphs is presented. The semantics are defined in a hierarchy that closely resembles the hierarchy of an arbitrary Rose RealTime model.

In [34] a translation of UML elements into the specification language CASL is proposed. The authors restrict translation to state machine constructs of which semantics are clearly defined. Another translation is defined in [38], the language chosen for the translation is PROMELA, of which models are checked with the model checker SPIN. Both attempts are interesting but show very little resemblance with the Petri net translation, which is why details are not presented here. The masters’ thesis of Daniel Ka Chung Yau [51], might be a good starting point for those interested in translations into other mathematical formalisms. The thesis itself presents a translation into the language BIR, the input language for the model checker BOGOR.
Rose RealTime

This chapter presents an introduction to Rose RealTime. It is started out with the presentation of some general information about the Rose RealTime, including information about the views that are used to model software systems. The main part will be about one of these views, the Logical View. Most of the information presented in this chapter is based on [32].

3.1 Background

The UML standard was constructed by the OMG from the different proposals suggested in response to the OMG Request For Proposal (RFP) of 1996. The first UML version, version 1.1, was adopted in 1997. Of great influence on the standard was the proposal submitted by the Rational Software Corporation, which put much effort in the definition of such a standard by combining different existing modeling approaches. In 1999 the OMG requested proposals for the definition of the UML 2.0 standard by a new RFP [30]:

“This RFP solicits proposals for a UML profile that defines standard paradigms of use for modeling of time-, schedulability-, and performance-related aspects of real-time systems.”

From 1997 had Rational Software Corporation been collaborating with ObjecTime Limited, the developer of the Real-Time Object-Oriented Modeling (ROOM) methodology. The tool ObjecTime Developer, which implemented ROOM, specifically aimed at the development of real-time operating systems.

The theories developed by ObjecTime were of great influence on the definition of the new version of the UML standard, which would not be finished until the end of 2004. Rational acquired ObjecTime at the beginning of 2000 and merged their software development solutions ObjecTime Developer and Rational Rose into Rose RealTime. The Rational Software Corporation was acquired by IBM in 2003, IBM incorporated Rose RealTime into their own product line. The implementation of UML in Rose RealTime is referred to as UML-RT.

The RFP boosted the development of tools supporting the requested paradigms. By requesting paradigms and not specifying one, it was inevitable that different tools using different paradigms were developed. The acceptance of the UML 2.0 standard confined this continuous divergence of viewpoints. However, by the end of 2004 had multiple paradigms been developed that can be regarded alternatives to UML-RT, two of these major alternatives will be discussed: RT-UML and RT-LOTOS.

RT-UML is supported in the Telelogic Rhapsody tool, developed by I-Logix [17]. A difference with UML-RT as supported by the Rose RealTime tool is that even the generated code is regarded a view of the system, the concept of a view will be explained in the sections yet to come. Regarding the generated code as a view is not necessarily an advantage or disadvantage, some developers
like the idea of having additional control over the code, while others like the idea of not being bothered with code details.

Another approach is taken with the development of RT-LOTOS [1]. RT-LOTOS is different from RT-UML or UML-RT in that it extends UML with mathematical constructs that enable a developer to formally specify timing constraints. It furthermore adds a tool to validate the created models.

3.2 Introduction

The Rose RealTime tool is developed by Rational, a division of IBM. Lately, the name of the tool has, with the arrival of a new version, been changed into IBM Rational Rose Technical Developer. The version known as Rose RealTime is used in this project and therefore the name Rose RealTime, or its abbreviation RoseRT, will be used in this document.

RoseRT enables developers to use a Model Driven Development (MDD) [42, 43] approach to the implementation of their software systems. As stated in [33]: “In MDD, the model is the application. Analysis, design, implementation, building, execution control, debugging and testing are all done at the model level”. For models created with RoseRT 90 percent of the underlying program code can be directly generated, the other 10 percent is specified by the developers in the model itself. Currently RoseRT enables the development of software systems in combination with C, C++, and Java programming environments.

As explained, is the Rose RealTime UML design formalism based on the ROOM methodology [11]. ROOM is an architecture description language, that is primarily applied in the telecommunications industry, and of which the concepts have proven themselves during their years of application.

The keyword RealTime is misleading in that the tool does not provide means for creating a system that adheres to predefined timing constraints. It is the task of the developer to implement this behavior using threads and priorities. As suggested in [15] a reason for the lack of timing constructs might be due to the fact that, in general, real-time systems with only soft timing constraint are used in the telecommunications industry.

The strength of the tool is further underlined by the announcement IBM made on January 19, 2007 [18]. It stated that the National Aeronautics and Space Administration (NASA) actively uses the IBM Rational software to develop the software systems that will operate the James Webb Space Telescope. Glen Cammarata gives a summarization of the capabilities and a motivation for the choice of the RoseRT tool in the announcement:

“It was important that NASA be forward-looking with the James Webb Space Telescope by using a systems development platform that would be reliable and ahead of the market throughout the extensive life of the mission. Since it’s based on open standards, the consistent and unified nature of the IBM software architecture helps curtail any problems that might present themselves down the road. Rational Rose RealTime software was the right choice for the critical nature
3.3.

of the James Webb Space Telescope. The software is much easier to manage and maintain pre and post launch.”

Models in RoseRT are developed from four different points of view, the use case view, the logical view, the component view, and the deployment view. The acknowledgment that a complex software system cannot be defined from a single point of view allows developers to thoroughly define their software systems from different angles.

The use case view is used to describe functional requirements of the software system. These are realized with, for instance, the use of message sequence charts and activity diagrams. The use case view gives an abstract description of intended system behavior without specifying the internal performance of tasks.

In the component view, classes, capsules, and protocols, all of which will be explained in the next section, are logically grouped together into components. A component physically packs model elements into executables or library files. A capsule that defines the entry point must be selected for every component.

Deployment modules are specified in the deployment view. Deployment modules are the actual instances of components. An instance might be different for different platforms, i.e., x86 versus Sparc architectures, which is why the deployment view is an extra view on top of the component view.

The logical view will be discussed in the next section. The logical view is actually the second view in the model. By some, although not specified as such, the four views are regarded as the four steps in the waterfall software development model: from functional requirements in the use case view, into analysis, design, and development in the logical view. Executable creation in the component view and finally system deployment in the deployment view.

3.3 Logical View

In this section an overview of the available UML constructs and how they are applied in the logical view is presented. First a short introduction explaining the main concepts is presented, followed by an explanation of the protocol and class constructs, and finally concluded by an explanation of the different aspects of the capsule construct.

3.3.1 Introduction

The relevant elements from the logical view are protocols, classes, and capsules. Protocols describe signals that can be sent to or received from ports defined on capsules. Classes are classes as known from object-oriented programming [28]. A special kind of classes called capsules are the constructs with which actual system behavior is modeled.

A RoseRT model is defined as a hierarchy of capsule instances, called capsule roles, encapsulated in one another. Encapsulation of capsule roles is defined in structure diagrams. A structure diagram defines, next to encapsulation, the
connections to other capsule roles. These connections are modeled with the use of ports, i.e. the constructs via which capsule roles communicate. Each port realizes a protocol. An instance of a port is referred to as a port role. Connections are created between port roles. Connections from a capsule role can be created to capsule roles on the same or one level higher or lower in the hierarchy.

An example of a structure diagram is depicted in Figure 3.1. The structure diagram models the interaction of a shop with its suppliers and customers. The different capsule roles are connected via ports, i.e. capsule role Customer is connected to capsule role Shop via ports PA and PB.

![Figure 3.1: Structure diagram example](image-url)

Each capsule has its own state diagram in which the dynamic behavior is defined. A state diagram consists of an initial point, junction points, choice points, composite and simple states, and transitions. The current state of a state machine is defined by the simple state it is currently in, the active state. A transition has a source state or point and a destination state or point. Transitions can be triggered and possibly execute code. A transition is triggered by the reception of a certain signal on a port and the code specified on the transition is atomically executed; its execution cannot be interrupted. A self-transition is a transition with the same source and destination state. Choice points have one incoming and two outgoing transitions, a Boolean predicate specifies which outgoing transition is executed. Transitions are connected to states using junction points. In a composite state, a state which is a state machine itself, junction points can be connected to a state or point within the composition.

Composite states enable the creation of a state hierarchy. When the active state is at a level $x$ in the hierarchy and a signal (trigger) is received a search for a transition to execute is started. First all outgoing transitions of the active state are examined, if none is found the search is continued at the composite state in which it is enclosed until the top state is reached. Transitions that originate on a composite state, and therefore can only be executed when one of the states it is composed of does not have a transition triggering on a certain incoming event, are referred to as border transitions.

State machines follow the run-to-completion paradigm, meaning that execution of a transition cannot be interrupted. Such a path from one simple state to another might contain multiple choice and junction points.
3.3. LOGICAL VIEW

Two examples of RoseRT state machines are depicted in Figure 3.2. Figure 3.2[a] depicts the state machine of capsule role Shop from Figure 3.1. Figure 3.2[b] depicts the state machine of composite state NotAvailable. In the upper left corner of both state machines the initial point is modeled. All rectangles, except NotAvailable depict simple states, the little dots on the borders of these rectangles depict junction points. The points Cancel and InStock depict choice points. Cancel models the decision of a customer to create a back order if an item is unavailable or to cancel the order. InStock models the action of inspecting whether an ordered product is currently in stock. From simple state Idle two transitions are modeled: CustomerOrder and BackOrder. Transition CustomerOrder is executed when an order is placed by a customer and initiates the inspection of the stock for the availability of the ordered item. When a product is received from a supplier the back order associated with this item is handled by transition BackOrder.

The run-to-completion paradigm can further be explained with choice point InStock. Execution of transition CustomerOrder is directly followed by the execution of either transition True or False originating at choice point InStock. If transition True is taken execution of the transition path ends when simple state DeliverOrder is reached, for transition False execution ends after the execution of transition InformCustomer.

![State machine of capsule role Shop](image)

**Figure 3.2: State machine of capsule role Shop**

### 3.3.2 Protocol

Communication in RoseRT models is realized via ports that exchange messages. The messages that can be sent to or received from a certain port are specified in a protocol. Every port realizes a protocol.

A protocol in RoseRT merely describes a collection of messages that can be sent to or received from a certain port realizing this protocol. Note how this differs from protocol definitions as known from for instance computer networks where it is normal that signal ordering and timing constraints are specified.

The messages that can be sent via a port realizing a protocol are modeled as a collection of signals, the protocols’ out signals. The messages that can be received are modeled as the protocols’ in signals. Every signal has a name,
which identifies the message throughout the system. Data can be attached to a signal being sent. The type of the attached data is determined at design time by associating a data class, classes will be explained in the next section, with a signal.

### 3.3.3 Class

Classes are classes as known from object-oriented (OO) programming paradigms \[28\]. They have attributes, operations and certain behavior. Classes are the blueprints of runtime instances called objects, of which the state is defined by the values of its attributes. The behavior of an object is defined by its operations, the results of operation invocations may be different depending on the state the object is in.

### 3.3.4 Capsule

Capsules are a specialization of classes, i.e. next to having attributes and operations can behavior be specified with structure and state diagrams. However, the semantics defined for a class are different from those defined for a capsule. Classes may have public operations, this is not possible for capsules, which use ports for interaction. The same applies to attributes: capsules do not have public attributes. The interface of a class is specified by its public attributes and operations, whilst the interface of capsule is defined by its ports. Another difference concerns communication, between classes communication is realized via public operation invocation, between capsules this is realized via public port message passing. The last major difference is in the manner behavior is defined: for a class behavior is defined by its operations, for a capsule it is defined in its state diagram. Table 3.1 summarizes the differences.

Classes mainly serve a supporting role in a RoseRT model. Class instances are, for instance, used as parameters (data) being sent with a signal over a certain port. They do not define communicating behavior in a RoseRT model.

<table>
<thead>
<tr>
<th></th>
<th>Class</th>
<th>Capsule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public attributes</td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Public operations</td>
<td>Allowed</td>
<td>Not allowed</td>
</tr>
<tr>
<td>Communication</td>
<td>Operation invocation</td>
<td>Port message passing</td>
</tr>
<tr>
<td>Behavior</td>
<td>Defined in operations</td>
<td>Defined in state diagram</td>
</tr>
<tr>
<td>Interface</td>
<td>Specified by public operations and attributes</td>
<td>Specified by ports</td>
</tr>
</tbody>
</table>

Table 3.1: Differences between classes and capsules

Capsules are the elements with which system behavior is modeled. Structure and state diagrams are the elements via which this behavior is specified. Struc-
ture diagrams define the capsule role hierarchy by encapsulating other capsule roles. Communication between the capsule role and its encapsulated capsule roles is defined in the structure diagram with the addition of ports and connections between them. State diagrams model the behavior of a capsule with the use of states and transitions. Transitions can be triggered by events received on ports created in the structure diagram, and might execute code that sends events via these ports to state diagrams in other capsule roles.

Details on how the combination of structure and state diagram models system behavior is presented in the next sections. Figures will be included to further clarify the RoseRT constructs. These figures are mostly taken from the coffee machine example that comes with the RoseRT tool. This example model is used throughout the entire document, the inner workings are not considered required knowledge for the coming sections and are explained in Chapter 8.

3.3.5 Structure diagram

Every capsule has a structure diagram in which communication with other capsule roles is modeled. Two example structure diagrams are shown in Figure 3.3, the top one is the structure diagram of capsule cuContainer encapsulated in the bottom structure diagram. RoseRT models are created top-down, which means that from a very abstract level the model is broken down into smaller parts to which more implementation details are, gradually, added.

This top-down approach is clearly shown in Figure 3.3, the design started out with the creation of a capsule drivers and a capsule cuContainer. To the structure diagram of the latter more implementation details are added by adding four more capsule to the system. Capsules that are placed inside a structure diagram are called capsule roles, the practice of placing capsule roles inside a structure is referred to as encapsulation. A capsule role is an instance of a capsule, i.e. drivers is an instance of BrewCycle_Test. A capsule might be instantiated more than once in a model, which enables the reuse of existing components. For a coffee machine with two plates for instance, an extra instance of CUWarmer could have been added.

Figure 3.3 actually shows the hierarchy of the entire coffee machine. Every model must have a top capsule, this is the capsule that is instantiated first and therefore specifies the system’s entry point, the encapsulated capsule roles are instantiated top-down.

At this point some definitions that are used in the remainder of the thesis will be introduced. Capsule roles that are encapsulated at the same level in the same capsule role are referred to as siblings, the capsule role encapsulating one or more other capsule roles is referred to as parent. The collection of capsule roles forming a path up the hierarchy, starting from the parent of a certain capsule role and including the top capsule, are called ascendants of this certain capsule role. Likewise, the capsule roles encapsulated in a certain capsule role are called children. The collection of capsule roles forming paths down the hierarchy from a certain capsule role, starting with the children, are called descendants.
CHAPTER 3. ROSE REALTIME

There are three types of capsule roles available. The type of a capsule role determines the manner in which a capsule role is instantiated. The simplest one is the fixed type capsule role, fixed type capsule roles are instantiated as soon as its encapsulating capsule role, its parent, is instantiated. Optional type capsule roles might be instantiated, and destroyed, somewhere in the lifespan of its parent via action code defined in the capsule’s state machine.

A plug-in type capsule role is instantiated by importing an already instantiated fixed or optional type capsule role. As with optional type capsule role the instantiation of plug-in type capsule roles is defined in state machine action code. In order to clarify the concept of importing capsule roles a parallel with object oriented (OO [28]) programming is drawn: importing a capsule role should be thought of as creating a reference to this capsule role. Which means that, as with OO programming, all actions being executed by the plug-in type
3.3. LOGICAL VIEW

capsule role are actually executed by the capsule role it is imported from. It is important to realize that therefore all plug-in type capsule roles share the active state with the capsule role they are imported from.

Cardinality is defined for every capsule role, being one by default. The cardinality property imposes an upper bound on the number of run-time instances of a capsule role. Thus, when a capsule role is specified to have a cardinality of three this means that at runtime at most three instances of the capsule role can be created. Recall the example that clarified the reuse of capsules, about the coffee machine with two plates. It was suggested that an extra capsule role would be added to the model. Another method is to define a cardinality of, for instance, two. By doing so the plate capsule role can be instantiated twice, which effectively achieves the same result.

For a fixed type capsule role the maximum number of instances is created upon instantiation of its parent. For optional and plug-in type capsule roles there are no requirements concerning the number of instantiations that should be created or the time they should be created, these decisions are left to the developer.

Next to exemplifying the hierarchy that exists in RoseRT models, does Figure 3.3 also shows some other constructs that will be explained in the upcoming sections, namely: ports and connectors.

Port

As shortly explained at the beginning of Section 3.3.4 a port is a modeling construct through which messages can be sent or received. A port belongs to a certain capsule, i.e. a port is instantiated when the capsule it belongs to is instantiated. Recall that every port realizes a certain protocol. The messages that can be sent via this port are specified in the protocols’ out signals, the ports’ out set, the messages that can be received are specified in the in signals, the ports’ in set.

The instance of a port is called a port role. Cardinality, one by default, is specified for every port. Port cardinality is used when connections to more than one port, possibly on different capsule roles, need to be realized. As with capsule role cardinality, it is not a requirement that the maximum number of port roles is instantiated. The property only imposes an upper bound on the number of port roles that can be instantiated.

When a port role has a cardinality greater as one the sending of signals becomes more complicated. A signal can either be broadcasted, i.e. send via all port roles, or it can be send via a specific port role. The difference in sending methods is defined by the C++ statements used to send a signal.

For two port roles pr1 and pr2 to be connected it is required that the out set of port pr1, i.e. out(Spr1) where Spr1 denotes the signal set of port pr1, is a subset of the in set of port pr2, in(Spr2), and vice versa, i.e. \( \forall s_1 \in \text{out}(S_{pr1}) \exists s_2 \in \text{in}(S_{pr2}) \) and \( \forall s_2 \in \text{out}(S_{pr2}) \exists s_1 \in \text{in}(S_{pr1}) \). When a port is defined to be conjugated it means that the protocol’s out signals become the ports’ in set, and the in signals become the ports’ out set. Using the conjugated property it is possible
to connect port roles using the same protocol. In Figure 3.4 numerous examples of conjugated port roles are depicted. The squares on the borders of a capsule role are the public port roles. The ones of which the name ends with a tilde sign (˜) are conjugated.

![Figure 3.4: Capsule structure diagram](image)

Two port types exist in RoseRT. End ports are the ports from which signals can be sent and on which signals can be received. Relay ports have connections to two port roles, signals received on one of them are forwarded to the other, hence the name relay port. Port role connections can only be created to port roles on capsule roles in the direct environment, i.e. to port roles on a parent, child or sibling. To create connections to ascendants or descendants at higher or deeper levels in the hierarchy relay ports are used. A connection path from one end port role to another end port role might contain an arbitrary number of relay port roles. Examples of relay ports are shown in Figure 3.3, where all ports on cuContainer ‘relay’ messages to or from encapsulated capsule roles.

In Figure 3.4 all ports on for instance capsule role sprayer are end ports. From Figure 3.3 it can be seen that the ports on the border of the structure diagram depicted in Figure 3.4 are relay ports, i.e. they originate on capsule role drivers and are relayed to one of the encapsulated capsule roles.

A port is either wired or unwired. Wired ports are explicitly connected using connectors. Unwired ports are dynamically connected at runtime. An unwired port is either published or unpublished. Unpublished ports, known as Service Access Providers (SAP), dynamically request connections to published ports at runtime. Published ports are known as Service Provisioning Points (SPP). The request for a connection is based on the name of the SPP. The port roles in Figure 3.4 are all wired, which can be derived from the fact that all the ports are explicitly connected at design time, i.e. in the model.

In this section it is explained how connections between port (roles) work and
what their properties are. The terms port role and port are used interchangeably for the sake of simplicity.

Messages can only be sent to or from a port when a connection with another port is realized. In RoseRT these connections are modeled with the addition of connectors to a structure diagram, a connector connects two ports. The lines between two ports in Figure 3.4 depict connectors.

Cardinality might be specified for a connector. This means that a certain connection can be instantiated a maximum number of times. Mostly, it is sufficient to specify port and capsule role cardinalities, connector cardinality can just be set to star (\(\ast\)), which means as many as needed, and will avoid conflicting cardinality properties. Four structural communication patterns can be identified from the combination of port, capsule and connector cardinalities.

The four structural communication patterns are depicted in Figure 3.5(a) through Figure 3.5(d). All four figures are composed of two parts, the upper part shows a compact RoseRT representation using the different cardinality properties. The lower part shows an alternative representation that clarifies the usage of cardinality properties in the upper part. In all examples does the upper part contain a capsule role on the left and one on the right, messages between them can be send in both directions. The cardinality property values of the different patterns are shown in Table 3.2.

Star

Figure 3.5(a) visualizes the star pattern. The left port role and the right capsule role have cardinalities greater than one. A capsule role with a cardinality greater than one is effectively the same as encapsulating the same capsule role multiple times in a structure diagram, thereby creating the star effect: a connector from the left port role is drawn to every capsule role on the right. That the cardinality of the right capsule role is greater than one can be seen by the number in the upper right corner, for port roles this is visualized by the addition of depth to the port role representation.

Bus

The bus pattern is depicted in Figure 3.5(b). The port roles on the left and right capsule role both have a cardinality greater than one, two in this case.

<table>
<thead>
<tr>
<th></th>
<th>Left</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Capsule Role</td>
<td>Port Role</td>
</tr>
<tr>
<td>Star</td>
<td>1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>Bus</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
<tr>
<td>Array</td>
<td>&gt; 1</td>
<td>1</td>
</tr>
<tr>
<td>Combination</td>
<td>&gt; 1</td>
<td>&gt; 1</td>
</tr>
</tbody>
</table>

Table 3.2: Communication pattern cardinality properties
The same effect can be modeled with the addition of two extra port roles, with cardinality one, to both capsule roles.

Figure 3.5: Structural communication patterns, if the cardinality of a capsule role is above one it is denoted in the upper right corner of the capsule role, for port roles this is depicted by adding depth to the square representing the port role.

Array

In the array pattern, which is depicted in Figure 3.5(c), both capsule roles have a cardinality greater than one: two in the depicted example. To model the same behavior without specifying cardinalities requires both capsule roles to be added an additional time, as depicted in the lower part of the figure.

Combination

As the name combination pattern suggests, is the last pattern actually a combination of the other three patterns. All elements have a cardinality greater than one. The right capsule role is added to the structure diagram four times,
3.3. LOGICAL VIEW

the left one two times. From both left capsule roles two connectors are drawn to the different right capsule roles. The pattern is visualized in Figure 3.5(d).

Note that in none of the pattern discussions the connector cardinality has been mentioned. This emphasizes the remark that usage of the connector cardinality property is not required.

3.3.6 State diagram

The behavior of a capsule is modeled in its state diagram. The state machine defined by the state diagram is started upon capsule role instantiation. Remember that for plug-in type capsule roles, as explained in section 3.3.3.1, the state machine is not started, it receives the state machine and its active state from the fixed or optional type capsule role from which it is imported. An example of a state diagram is depicted in Figure 3.6(a), it is placed in context with the structure diagram in Figure 3.6(b).

The elements from which a state diagram is modeled are: an initial point, zero or more junction points, zero or more choice points, zero or more states, and the transitions connecting the previous four elements.

As with structure diagrams, state diagrams are hierarchies of states starting with the top state. States that enclose other states are called composite states. The state diagram in Figure 3.6(a) actually is a composite state, it encloses 4 other states. In this case the enclosed states itself do not enclose other states, therefore, they are referred to as simple states.

Although a structure diagram can be modeled without any knowledge about the state diagram, the exact opposite is true for a state diagram. To enable behavioral modeling, interaction with other parts of the system is required, i.e. signals may need to be sent to or received from other parts in the system. These signals are sent or received via ports defined in the structure diagram. The elements in a state diagram, therefore, rely on the elements in the structure diagram. The context in which one should think about a state diagram is depicted in Figure 3.6(b): the state diagram can and should not be examined without the structure diagram.

A transition defines a relation between two, possibly the same, states in a state transition diagram: the source and destination state. A transition might be triggered, which means that a certain event must occur prior to transition execution. An event is the reception of a certain signal on a certain port. So for instance in Figure 3.6(a) it might be the case that transition Brew triggers on a signal received on port boiler.

For every transition in the state diagram action code might be defined. Action code is arbitrary programming code, this code can inhibit virtually every statement: e.g. port connection operations, capsule role instantiation, and the sending of messages, it is only limited by the expressive power of the chosen programming language. C++ programming code will be examined for the translation of RoseRT models to Petri nets.

The next sections will discuss the previously mentioned state diagram ele-
ments. The first element to be discussed is the initial point, followed by the choice point, the simple and composite state, the junction point, and finally the transition.

**Initial point**

A state machine is started upon capsule role instantiation by taking, if modeled, the initial transition. The initial transition originates on the initial point and terminates on one of the other points or states. Every composite state has an initial point. When in the top state no transition from it is defined, the state machine will not be started. Not starting a state machine is useful when a capsule role is only added to create an extra hierarchical level, i.e. `cuContainer` in Figure 3.3.

Obviously, triggers can not be defined for initial transitions. Suppose a capsule role hierarchy in which all encapsulated capsule roles are of the type fixed. These are all instantiated at the same time. If all initial transition would be triggered this would most certainly lead to a deadlock, which is why triggers
can not be defined for initial transitions.

The initial transition is often used to initialize the capsule: e.g. the instantiation of encapsulated optional or plug-in type capsule roles. An initial point is visualized by a single dot in the state diagram, for instance the initial point in Figure 3.6(a) has a transition to simple state NotBrewing.

**Choice point**

A choice point has at least one incoming transition and precisely two outgoing transitions. Which outgoing transition is taken depends on the condition that is defined in the choice point. This condition is formulated as a Boolean predicate. The fact that it is a Boolean predicate justifies the choice for two outgoing transitions; one, the true transition, is taken upon a true evaluation, the other, the false transition, upon an evaluation to false. An example is shown in Figure 3.7.

As soon as a transition terminates on a choice point the Boolean predicate is evaluated and either the true or false transition is immediately taken. As defining triggers on the outgoing arcs might contradict with the outcome of the predicate this is strictly prohibited.

**Simple state**

Simple states are states that do not enclose any other states or points. In the following definition is given: “A state is the condition during the life time of an object where it is ready to process events”, meaning that the state of a state machine is defined as the simple state it is currently in; the active state. Processing events should be interpreted as executing a transition that triggered on the reception of a certain signal.

For a state, including the composite one, entry and exit actions might be defined. These are, like the action code for transitions, normal code blocks. Entry actions are executed when a state is entered, i.e. a transition terminates on a simple state, or enters a composite state. Exit actions are executed when a transition from a state is taken.
Figure 3.8 depicts some examples of simple states, namely: SA, SC, SK, and SL.

**Composite State**

A composite state is a state that is composed of the same elements a state diagram is composed of. Two examples of composite states are shown in Figure 3.8: SB and the top state, which is the top part of the figure. Both have their own initial point, states and transitions.

Note that in this case, the state machine of composite state SB is not started by taking the initial transition, there is none, but rather by an incoming transition. In the example it is started with the incoming transition $u$ followed by $w$ which eventually terminates on state SK.

**Junction point**

The previous sections discussed, amongst others, the origination and termination of transitions on states. However, a transition never enters or exits a state directly, this is realized via junction points. A junction point is an element defined on a state, both composite and simple, used to connect transitions to this state.

Junction points are always located on or attached to the border of a state, examples are, again, shown in Figure 3.8: the dots on the borders of the simple states on which transitions originate and terminate depict junction points as well as the dots connected to the borders with arrows. In the top state $u$ terminates on a junction point on composite state SB. Inside this composite state another transition, $w$, from this junction point is taken that terminates on state SK.
A junction point can have an arbitrary number of incoming transitions but only one outgoing transition. When no outgoing transitions are modeled the junction point is referred to as a terminating junction point. Junction points on simple states are, obviously, always terminating junction points. Like with choice points the outgoing transition of a junction point is immediately taken and can not be triggered. In an abstract view a junction point can be compared to a port, the end port with the terminating junction point, the relay port with the non-terminating junction point.

The initial transition of a composite state is taken upon termination of a transition on one of its terminating junction points, thereby initializing the state machine.

**Transition**

The beginning of section Section 3.3.6 introduced the transition element. In summary: transitions originate and terminate on points in the state diagram and might be triggered by the reception of a certain signal on a certain port. Note how it is stated that transitions originate and terminate on points, not on states. It is explained in the discussion on junction points that termination on a simple state is in fact termination on a junction point resulting in the active state becoming the simple state it is on. Although this summarizes the main use of transitions, a more elaborate discussion is required and presented in this section.

Multiple triggers might be defined for a transition, i.e. it is possible that a transition triggers on the reception of either signal $A$ or signal $B$ on a certain port. Or even on different signals from different ports. It is enough to receive one of the signals in a transitions’ trigger set to initiate its execution.

A transition path is a trail of transitions from one simple state to another, possibly entering and exiting multiple composite states. Such a path may contain an arbitrary number of choice and junction points. In a runtime environment one does not refer to the execution of transitions, but rather to the execution of transition paths. The execution of such a path is initiated by the reception of a signal on which the first transition on the path triggers. Once execution of a transition path is started it cannot be interrupted by the reception of other signals, i.e. at all time at most one path per capsule role instance is being executed. In other words: transition paths are being executed as atomic actions. This method of execution is referred to as run-to-completion. A further clarification might be found in Figure 3.9, which contains two possible transition paths: from state $SA$ to state $SB$ or state $SC$. Both are executed as one atomic action, executing action code defined on the different transitions in sequence.

An implication of the run-to-completion paradigm is that only the first transition can be triggered, if there would be other triggered transitions somewhere on the execution path this would contradict the restriction that execution can not be interrupted by the reception of a signal. It immediately follows from the definition of choice and junction points that in RoseRT only a transition from a simple state can be triggered, which confirms the implicit restriction.
RoseRT applies a set of transition selection rules for the determination of the transition to be taken upon reception of a certain event. The search for a transition is started from the active state. It is evaluated whether any of the outgoing transitions of the active state triggers on the received signal, if so this transition is taken. If not the search for a transition is repeated on the composite state one level higher up the hierarchy. This process is repeated until the top state is reached or until a transition is found. With the exception of certain self-transitions, which will be discussed next, a transition is only executed after the execution of all exit codes on composite states that might have been left during the search up the hierarchy.

Next to regular transitions, originating on a state and terminating on another state, it is also possible to model self-transitions. Self-transitions are transitions that have the same source and destination state. There are three types of self-transitions available in RoseRT. Differences between the self-transition types are in which entry and exit codes are executed and in how the next active state is selected. A summary of the differences is denoted in Table 3.3.

- **External self-transition**: Execution of this transition results in exiting and re-entering the state it is defined on, and, therefore, execution of the entry and exit actions. Note that first the exit action, then the transition action, and finally the entry action code is executed. If this transition is defined on a simple state the active state will remain the same, i.e. this simple state. If it is defined on a composite state...
the next active state is determined by how the composite state is initialized.

Inner self-transitions behave the same way as external self-transitions do, except that the entry and exit codes for the state it is modeled on are not executed, hence the keyword inner. An example is depicted by transition $u$ in Figure 3.10(b). Figure 3.10(b) depicts an alternative representation of a junction point: a circle connected to the border with $H^*$ in it.

The last one is the inner internal self-transition. No states are left or entered upon execution of this type of self-transition, which implies that no action code is executed. The active state, therefore, remains the same. The only executed code is the action code that is defined on the transition. Transition $u$ in Figure 3.10(c) depicts an example of this type of self-transition.

The external self-transition is the only type of self-transitions that can be modeled on a simple state, i.e. inner and inner internal self-transitions can only be modeled on composite states.

<table>
<thead>
<tr>
<th>self-transition</th>
<th>action code execution</th>
<th>state change</th>
</tr>
</thead>
<tbody>
<tr>
<td>external</td>
<td>including entry and exit action of state it is modeled on</td>
<td>possibly if attached to composite state, no if otherwise</td>
</tr>
<tr>
<td>inner</td>
<td>not including entry and exit actions of state it is modeled on</td>
<td>possibly if attached to composite state, no if otherwise</td>
</tr>
<tr>
<td>inner internal</td>
<td>only transition action code</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 3.3: Self-transition comparison

In this document self-transitions modeled on composite states are referred to as border transitions. Thus, border transitions can have three types: external, inner, or inner-internal. Self-transitions on composite states are not the only transitions that are referred to as border transitions. Transitions not being self-transitions originating on a composite state, i.e. that originate on a junction point to which no incoming transitions are modeled, are also referred to as border transitions. It is assumed that border transitions are always triggered, system behavior would be very unpredictable if otherwise.

3.4 Event handling

Every capsule role is instantiated with an empty message queue in which messages sent to this capsule role are stored. Upon reception of a message it is placed at the end of the message queue. The next event that will be processed by RoseRT is the event at the front of the queue. If no transition is found that triggers on this message it is discarded. Via this queue run-to-completion is secured: messages received while executing a certain transition path are placed at the end of the queue until the capsule role is ready to process the next event, i.e. when the capsule role’s state machine has entered the next simple state.
Petri net preliminaries

This chapter presents an introduction to the theory of Petri nets, as well as a description of the Petri net analysis tool used for the verification of RoseRT models. An introduction to the different analysis methods that can be performed is presented in the last section.

4.1 Petri net introduction

As the translation of RoseRT models into Petri nets is mostly defined in a mix of textual and graphical definitions is a very formal introduction omitted. There are some very good and standard publications that have perfect and thorough formal definitions of Petri nets, those interested in such formal definitions should refer to for instance [35].

Petri nets are presented graphically with the use of places, tokens, transitions and arcs. A place is depicted as a circle, a token as dot, a transition as a rectangle and an arc as a solid line with one arrowhead between a transition and a place. An arc is never drawn between two transitions or two places.

A token is drawn inside a place, a place can contain multiple tokens. A place is said to be marked when it contains tokens, its marking is defined by the number of tokens it contains. The collection of currently marked places defines the state of the system.

The system state changes with the firing of a transition. A transition can fire, i.e. is enabled, if all the places connected to its input arcs contain at least one token. When a transition fires it outputs a token to every of the places connected to its output arcs. The firing of a transition is an atomic event, i.e. it cannot be interrupted by the firing of other transitions.

The Petri nets in the class discussed so far are known as Classic Petri nets, numerous other classes such as Colored Petri nets, Timed Petri nets, Stochastic Petri nets, Object Petri nets, etc. exist. The translation presented in this paper is defined using Classic Petri nets.

An example

Figure 4.1 depicts twice the same Petri net representation of two communicating traffic lights; when one traffic light is green or orange the other must be red. Figure 4.1(a) depicts the state of the Petri net in which both traffic lights are red, it should be possible for only one of them to turn green. Figure 4.1(b) confirms this as the token from place atomicity is taken, which means that transition rg1 is no longer enabled.

The token is eventually returned to place atomicity when transition or2 fires. When the token is returned to place atomicity both traffic lights, but no more than one, can turn to green.
4.2 Petri net tools

This section discusses the tools that are used for creating and analyzing Petri nets. Many Petri net tools use their own proprietary format for storing Petri nets. With possible future extensions in mind, it was tried to search for a format that is or has the potential to become a format that is accepted as the Petri net standard.

Such a standard has been developed and is named Petri net Markup Language (PNML). The PNML standard is XML based and defines a format that allows for the description of the most common Petri net constructs. Furthermore, developers are allowed to add tool-specific features to their own PNML dialect.

It was chosen to use a PNML dialect, namely EPNML [49], to store translated RoseRT models. EPNML is used by Yasper, which is an acronym for Yet Another Smart Process Editor [50]. Yasper is a Petri net editor that allows for the creation and simulation of Petri nets and is developed at the Eindhoven University of Technology. Many libraries have been developed with Yasper that abstract over the XML syntax of an EPNML file. These libraries are integrated into the tool created for the analysis of RoseRT models.

![Figure 4.1: Petri net model of two communicating traffic lights](image)

The number of analysis tools is probably even bigger than the number of different Petri net storage formats. The tool LoLa, Low Level Petri net Analyzer [39, 40], is used to perform Petri net analysis tasks. It will first be explained why this tool is chosen, followed by a short list of the other options.

LoLa was specifically developed without a graphical interface to allow for other tools to easily integrate Petri net verification procedures. LoLa is developed as not being a stand-alone tool that offers numerous analysis options; it has to be compiled with different settings for every specific analysis task. Requiring compilation of LoLa for every analysis task allows for the creation of executables that are optimized for exactly this task. For instance, in the project as presented in this thesis two LoLa instances have been integrated: one for deadlock analyzes, and one for the verification of CTL formulas.
4.3. ANALYSIS METHODS

LoLa, as most other Petri net analysis tools, has its own proprietary input format. The Yasper libraries contain the functionality to convert PNML files from and to the LoLa format.

Other analysis tools that have been investigated are for instance the Time Petri Net Analyzer (TINA) and the Integrated Net Analyzer (INA). TINA has been eliminated as an option since even within the tool additional format conversions have to be performed for different analysis tasks, which makes the process of analyzing Petri nets unnecessarily complicated. INA is not used as it is currently outdated by LoLa, INA served as the basis for the development of LoLa.

4.3 Analysis methods

Two of the available LoLa analysis tasks have been incorporated: deadlock analysis and Computational Tree Logic (CTL) formula verification. By incorporated it is meant: integrated in the RoseRT analysis toolkit as developed during the research period.

The selection of analysis tasks is made in close consideration with the RoseRT developers at Océ-Technologies B.V. They proposed a series of questions that they would like to have answered about their model. The most common questions have been translated into analysis tasks.

Deadlock analysis

Deadlock analysis is used to verify the absence of deadlocks in a system. A deadlock is a Petri net state in which no transitions are enabled, i.e. there is no transition that can fire, which means that the system is ‘dead’: it will remain indefinitely in the same state. Normally, in real-time systems, this is regarded undesired behavior, a system should always be able to process new events. Deadlock analysis will be referred to as analysis task \textit{AD1} throughout this paper.

Figure 4.2 depicts the two traffic lights in a deadlock situation. When no token is available in place \textit{atomicity} no light will be able to turn from red to green.

CTL formula verification

CTL formulas allow for the verification of the existence or absence of certain paths through a Petri net. A short introduction to CTL is presented here, for details on CTL refer to [3].

The CTL grammar is denoted in Grammar (4.1), where \( p \) represents an atomic formula, \( A \) means along all paths, \( E \) means along at least one path. \( A \) and \( E \) are referred to as path operators.

The \textit{neXt}, \textit{Globally}, \textit{Finally}, and \textit{U}ntil operators are known as state operators. \( X\phi \) means that \( \phi \) must hold in the next state, \( G\phi \) that \( \phi \) must hold on the
entire subsequent path, $\phi$ must eventually hold somewhere on the subsequent path for $F\phi$, and $\phi U \psi$ specifies that $\phi$ holds until somewhere on the subsequent path $\psi$ holds with the restriction that $\psi$ must eventually hold. The mapping of CTL elements onto the LoLa CTL input language is denoted in Table 4.1.

$$
\phi ::= \bot | \top | p | (\neg \phi) | (\phi \land \phi) | (\phi \lor \phi) | (\phi \rightarrow \phi) | \\
AX\phi | EX\phi | AF\phi | EF\phi | AG\phi | EG\phi | \\
A[\phi U \phi] | E[\phi U \phi]$$

(4.1)

CTL formula verification is normally referred to as model checking. For the toolkit has LoLa model checking been integrated in three variants, AMC1, AMC2, and AMC3. The variants only differ in the CTL formula to be verified, the LoLa instance is equivalent.

A CTL formula to verify the absence of deadlocks can very easily be constructed. It is chosen not to do so as a version of LoLa that is optimized especially for deadlock analysis can be constructed.

AMC1

The first variant, AMC1, verifies whether a marked place will always, i.e. over all subsequent paths, become marked again. Such a check is often needed in
real-time systems as these mostly define some *idle* state from which it is possible to perform a task. After execution of a task the system should always return to this *idle* state and wait for an event that starts a new task. The generic LoLa CTL format for analysis method AMC1 is denoted in CTL Formula (4.2), where $p$ represents a random place in a Petri net.

\[
\text{ALLPATH ALWAYS ALLPATH EVENTUALLY} (p > 0) \quad (4.2)
\]

Figure 4.3 depicts two, very small, example Petri nets. Execution of AMC1 for place $pl2$ would result in a *false* evaluation for Figure 4.3(a), as place $pl2$ enables transition $tr2$ after which place $pl3$ will be marked. Place $pl3$ does not enable any transitions, i.e. it is not possible for place $pl2$ to become marked again. For Figure 4.3(b) will evaluate to *true* for an AMC1 analysis of place $pl2$. When place $pl2$ is marked it is always possible to become marked again, i.e. via transition sequence $tr2, tr3$.

![Petri nets](image)

(a) (b)

Figure 4.3: Two examples of very simple Petri nets

**AMC2**

Variant AMC2 is used to verify that from a certain marked place, $p1$, it is always possible to reach another place, $p2$. For many real-time systems it is important to verify the system startup process, i.e. when place *booting* is marked does it always result in place *idle* being marked. The LoLa CTL format is denoted in CTL Formula (4.3), in which the implication is removed as LoLa does not know the construct. With implication the formula is written as denoted in CTL Formula (4.4).

\[
\begin{align*}
\text{ALLPATH ALWAYS} & \ ( ( p1 = 0 ) \ OR \ \text{ALLPATH EVENTUALLY} ( p2 > 0 ) ) \\
\text{ALLPATH ALWAYS} & \ ( ( p1 > 0 ) \rightarrow \ \text{ALLPATH EVENTUALLY} ( p2 > 0 ) )
\end{align*}
\quad (4.3)
\]

\[
\begin{align*}
\text{ALLPATH ALWAYS} & \ ( ( p1 = 0 ) OR \ \text{ALLPATH EVENTUALLY} ( p2 > 0 ) ) \\
\text{ALLPATH ALWAYS} & \ ( ( p1 > 0 ) \rightarrow \ \text{ALLPATH EVENTUALLY} ( p2 > 0 ) )
\end{align*}
\quad (4.4)
\]

For both examples in Figure 4.3 would an AMC2 analysis with $pl1$ as $p1$ and $pl3$ as $p2$ evaluate to *true*, as a path from place $pl1$ to place $pl3$ always exists. An AMC2 analysis with $pl3$ as $p1$ and $pl2$ as $p2$ would not be evaluate to *true* in Figure 4.3(a), as from place $pl3$ no transitions can be taken.
AMC3

Analysis method AMC3 is very similar to AMC2, it also checks for the existence of a path from one marked place to another. For AMC3, however, two end places can be defined. It is verified whether from a certain marked place it is always possible for one out of two defined end places to become marked. The CTL notation is denoted in CTL Formula (4.5), where $p1$ is the start state and $p2$ and $p3$ the end states.

\[
\text{ALLPATH ALWAYS } ( ( p1 = 0 ) \text{ OR ALLPATH EVENTUALLY } ( ( p2 > 0 ) \text{ OR } ( p3 > 0 ) ) )
\] (4.5)

In the example of a real-time system given for AMC2 it was assumed that the boot process would always end in an idle state. However, often there is an option for a system to boot successfully or unsuccessfully, i.e. ending up in idle or error. In these cases should analysis method AMC3 be used.
The translation of Rose RealTime models into Petri nets is presented in this chapter. Section 5.1 presents RoseRT usage guidelines that, if adhered to, allow for a correct translation into Petri nets. The translation is presented in Section 5.2 starting with the translation of structural elements into Petri nets in Section 5.2.1, followed by the translation of behavioral elements to Petri nets in Section 5.2.2 and finally the translation of C++ code fragments to Petri nets in Section 5.2.3. The presentation of the translation is followed by Section 5.3 in which limitations of the translation are presented and explained. At last, does Section 5.4 present the translation of Petri net analysis results back into RoseRT models.

5.1 Usage guidelines

The translation presented in this chapter is defined as generic as possible. However, some assumptions on the usage of RoseRT constructs need to be made and are presented here.

Unique identifiers

The RoseRT tool can generate unique identifiers for every element in a model. It is assumed that these identifiers are generated before the creation of a Petri net. The identifiers are used to (1) generate unique place identifiers in the Petri net and (2) to translate verification results back into RoseRT.

Asynchronous communication

The Océ coding guidelines strictly prohibit the use of synchronous message passing. The state machine from which a synchronous message is sent is blocked, i.e. does not perform any actions, until a certain reply signal is received. It is not known, without detailed code analysis, what signal the state machine is waiting for. To translate synchronous message passing it would require the addition of synchronization transitions and places for every possible signal being sent to the capsule role.

Being unable to feasibly determine what signal communication synchronizes on together with the fact that the Océ models do not contain synchronous message passing makes it that the current translation is only applicable to models using asynchronous message passing.

Although not detailed in this thesis are some hints on how the translation of synchronous message passing could be implemented presented here. Suppose that transition $tr1$ in Figure 5.1 sends a synchronous message to capsule role $B$. Blocking of capsule role $A$ until an answer is received is modeled with place $pl3$. Capsule role $A$ continues execution when place $pl2$, the synchronization place, becomes marked.
5.2 Model translation

The translation of a RoseRT model is divided into three parts: the structural, the behavioral, and the C++ code translation. In total 13 patterns have been formulated to translate the different constructs.

First and foremost the implications of capsule role cardinality need to be explained. A capsule role, and thus its structure and state diagram, is translated the number of times defined by its cardinality.

Second, a short introduction about code analysis is required before the discussion of the structural and behavioral translation patterns. Code analysis is restricted to structural and design-time code analysis, which means that variable values are ignored. This implies that all runtime actions depending on variable values are not identified as such, i.e. an if statement is recognized, but not the values on which the evaluation is based. It is analyzed whether a signal is sent from within a selection, iteration, or sequential context. Note that these three may be nested, e.g. an if statement may contain multiple while statements.

5.2.1 Structural translation

This section discusses the structural patterns. These are concerned with the translation of capsule roles, ports and the connections between them. All translation are explained with figures from RoseRT and their Petri net translations. The ports shown in the RoseRT figures all realize the same protocol, this protocol has in signals $A$ and $B$ and out signals $Y$ and $Z$.

This section consists of two parts, the first is concerned with the presentation of the structural patterns, the second with the presentation of some means with which statistical data of the translated models can be calculated.
5.2. MODEL TRANSLATION

Pattern: SD01

This pattern is used to create Petri net places for a port on a capsule role.

For every port \( p \in P_{cr} \), where \( P_{cr} \) denotes the set of ports modeled on a certain capsule role \( cr \), the set of signals \( S_p \) that can be either sent to or received from port \( p \) is defined by its protocol. The signals that can be sent are denoted as \( out(S_p) \), the signals that can be received as \( in(S_p) \), where \( S_p = out(S_p) \cup in(S_p) \).

For every signal \( s \in S_p \), a uniquely labeled, place is created. The set of unique labels, and thus the set of places, is defined by the union of Formula (5.1) and Formula (5.2), where \( c(\ldots) \) denotes the cardinality of a port or capsule role.

\[
\bigcup_{i=1}^{c_{cr}} \bigcup_{p \in P_{cr}} \bigcup_{j=1}^{c_p} \bigcup_{s \in out(S_p)} cr.i.p.j.s.out \quad (5.1)
\]

\[
\bigcup_{i=1}^{c_{cr}} \bigcup_{p \in P_{cr}} \bigcup_{j=1}^{c_p} \bigcup_{s \in in(S_p)} cr.i.p.j.s.in \quad (5.2)
\]

Every label ends with either \( .in \) or \( .out \). This postfix is appended to be able to distinguish between the sending and receiving of signals.

The structure diagram depicted in Figure 5.2(a) shows a port on the border of a capsule role. This port realizes the example protocol, which means that only two signals can be sent and only two can be received. The Petri net translation is depicted in Figure 5.2(b). There are no capsule or port roles with a cardinality greater than one.

![Figure 5.2: Pattern SD01, port \( pA \) on capsule role \( cr \) realizes the example protocol which means that signals \( A \) and \( B \) can be received and \( Y \) and \( Z \) be sent.](image)

Pattern: SD02

Pattern SD01 denotes the translation of ports and signals for a capsule role. Pattern SD02 adds the connections between different capsule role translations, i.e. between the places representing the port roles. The translation of connections presented here is slightly based on the translation as presented in [10].

For every two connected port roles \( pr1 \) and \( pr2 \in PR \), i.e. \( connected(pr1, pr2) \equiv True \land pr1 \neq pr2 \), where \( PR \) denotes the entire set of port roles in the RoseRT model, transitions are created between the places representing them. The places representing out signals, i.e. places that end with \( .out \), are connected via a transition with places representing in signals.

Figure 5.3(a) depicts two capsule roles with one connection between them. Note that port \( pA \) on capsule role \( client \) is a conjugated port, which means that its
in signals are the out signals of capsule role server. Figure 5.3(b) depicts the Petri net translation. All elements in Figure 5.3(a) have a cardinality of one.

The communication patterns star, bus and array, as discussed in Section 3.3.5, with cardinalities greater than one are also translated with the application of patterns SD01 and SD02, the results are depicted in Figure 5.4(a) through (f).

Figure 5.4(a) depicts a RoseRT structure diagram with two encapsulated capsule roles that are interconnected using the star pattern, i.e. multiple instances of capsule role server are all connected to different port roles of capsule role client. The Petri net translation is shown in Figure 5.4(d), in which the numbers \(i\) and \(j\) from Formula (5.1) and Formula (5.2) clearly show the influence.
of capsule and port role cardinalities on place creation. Signals B and Y are left out of the translation for reasons of simplicity.

Figure 5.4(b) presents the RoseRT view of the bus communication pattern, its translation is shown in Figure 5.4(c). The differences in cardinality in the RoseRT structure diagrams depicted in Figure 5.4(a) through (c) may not be immediately visible, therefore, some clarification is presented here. When a capsule role has a cardinality that is greater than one, this number is shown in the upper right corner of this capsule role, furthermore the borders of the capsule role have more depth. For a port role the addition of depth to its representation is the only visible element enabling the identification of its cardinality being greater than one.

The translation of the array communication pattern is depicted in Figure 5.4(c) and Figure 5.4(f). No translation is presented for the last communication pattern, the combination communication pattern, as it is merely a combination of the other three patterns.

Statistics

It is important to have some means of calculating lower bounds on the number of places that will be created in the Petri net. In this section the different formulas to calculate the number of places needed to represent the port instances in the model are given.

Formula 5.3 is used to calculate the number of places needed to translate a port role, where \( PR \) denotes the set of port roles in the model, \( signals(...) \) the set of signals for a certain port role, and \( c(...) \) the cardinality of a port or capsule role. Formula 5.4 is used to recursively calculate a lower bound on the total number of places needed for port role translations of a capsule role hierarchy. \( CR \) denotes the set of capsule roles in a model, \( PR_{cr} \) the port roles of capsule role \( cr \), and \( ecrs(...) \) a capsule role’s set of encapsulated capsule roles.

\[
\forall pr \in PR : NrPortPlaces(pr) = |signals(pr)| \cdot c(pr) \quad (5.3)
\]

\[
\forall cr \in CR : NrPortPlaces(cr) = \\
c(cr) \cdot \sum_{pr \in PR_{cr}} NrPortPlaces(pr) + \\
\sum_{ecr \in ecrs(cr)} NrPortPlaces(ecr) \quad (5.4)
\]

Formula 5.5 is used for the calculation of the number of transitions, where \( connected(...) \) denotes a Boolean condition evaluating to true when its parameters are connected and \( p1 \neq p2 \).

\[
\forall pr_1, pr_2 \in PR : NrConnectionTransitions() = \\
|connected(pr_1, pr_2)| \quad (5.5)
\]
5.2.2 Behavioral translation

This section describes the patterns for the translation of the behavioral elements of a RoseRT model. The patterns are shown in their most simple form, concerns like for instance transition atomicity and message queues are initially not taken into account.

The behavioral patterns are divided in six pattern groups: state and point translations (prefixed by SP), regular transitions (RT), capsule role initialization (CI), border transitions (BT), composite states (CS), and choice points (CP).

Context

Discussion of the behavioral patterns requires clarity on the context these patterns are created in. Figure 5.5(a) and Figure 5.5(b) depict the overviews from both the RoseRT and the Petri net perspective; the clouds represent the state machines in the RoseRT view and their translations in the Petri net view. In the RoseRT view the port roles are visualized as squares on the borders of the capsule roles, in the Petri net these are translated into the places outside the clouds. The connections, visualized in RoseRT using lines between port roles are translated using transitions between the places representing the port roles. The state machines in RoseRT are placed inside the capsule roles to emphasize their relation with the port roles. The arcs going from and to the places outside the clouds in the Petri net visualize the dependence of the state machine translation on the places representing the port roles.

Figure 5.5: Contextual relationship between structural and behavioral translation
Pattern SP01

The places representing the relevant states and points in a state diagram are created with pattern SP01. For all the simple states modeled in the state diagram a place in the Petri net is created. The set of labels created for the translation of the simple states in a capsule role $cr$ is defined by Formula (5.6), where $c(\ldots)$ denotes the cardinality of the capsule role, and $ss(s\ldots)$ the set of simple states contained in the capsule roles’ state diagram.

\[ \bigcup_{i=1}^{c(cr)} \bigcup_{ss \in ss(s(cr))} cr.i.ss \]  

(5.6)

The initial point is translated into a Petri net place, labeled the same way as a simple state. Choice points in the state diagram are also translated, the set of labels for them is given in Formula (5.7), where $cps(\ldots)$ denotes the set of choice points in a capsule role $cr$.

\[ \bigcup_{i=1}^{c(cr)} \bigcup_{cp \in cps(cr)} cr.i.cp \]  

(5.7)

Junction points on simple states are not translated they are directly mapped onto the simple state on which they are defined. Junction points on composite states are translated as places in the Petri net as they are required to model a transition paths with. The set of junction point labels is given in Formula (5.8), where the set of composite states in a capsule role is denoted by $css(\ldots)$ and the set of junction points on a composite state by $jps(\ldots)$.

\[ \bigcup_{i=1}^{c(cr)} \bigcup_{cs \in css(cr)} \bigcup_{jp \in jps(cs)} cr.i.jp \]  

(5.8)

Pattern RT01

Pattern RT01 describes the translation of a transition $t$ between two simple states $SA$ and $SB$ in a state machine. Remember that a transition between two simple states, as explained in Section 3.3.6, is actually a transition between two junction points on the borders of these simple states. Transition $t$ might be triggered by a set of triggers $E$ and, furthermore, a bag of signals $S$ might be generated by the execution of its action code $c$.

For each trigger $e \in E$ a transition between $SA$ and $SB$ is created that is enabled with the signal defined by $e$, and which places tokens in the places representing the signals in $S$. A minimal example translation is depicted in Figure 5.6. The translation of events as proposed in [10] is like the translation defined here.

Transition $t$ in Figure 5.6 triggers on signal $A$ received on port $pA$, signal $Z$ over the same port is sent during execution. The Petri net translation is shown in Figure 5.6. Transition $t$ is placed between places $SA$ and $SB$, representing the two states. It has an additional input place $pA.A$, modeling the trigger, and an additional output place $pA.Z$, modeling the signal being sent.
Figure 5.6: Pattern RT01, the RoseRT notation has been altered: extra information is added to the transition label. The format is: \((A|B)\), where A denotes the set of signals the transition triggers on and B the set of signals that is generated.

Figure 5.7: Pattern RT01, example applications

Four example applications of RT01 are depicted in Figure 5.7. Pattern RT01-1, Figure 5.7(a) and (b), is applied in the case that transition \(t\) is not triggered and does not generate any new events, i.e. the bag of signals being sent remains empty. Pattern RT01-2, of which the RoseRT representation is depicted in Figure 5.7(c) and its translation in Figure 5.7(d), is applied when a transition is triggered but does not produce new events. RT01-3, original and its translation depicted in Figure 5.7(e) and (f) respectively, can be regarded the counterpart of RT01-2; transition \(t\) is not triggered but does generate events.

RT01-MT is the case in which multiple triggers, depicted in Figure 5.7(g), are defined for transition \(t\). As RT01 prescribes does the translation in Figure 5.7(h) have Petri net transitions for every trigger. Execution of one of these transitions results in generation of new event, i.e. after execution a token is in place \(p.A.Z\).

Pattern RT01 describes the translation of a transition between two simple states \(SA\) and \(SB\). The definition does not define restrictions on \(SA\) and \(SB\), which makes the pattern applicable to self-transitions, i.e. where \(SA = SB\). An example of a RoseRT self-transition is depicted in Figure 5.8(a), its translation in Figure 5.8(b). The example in Figure 5.8(c) is a self-transition with multiple triggers, i.e. \(E = \{p.A.A, p.A.B\}\), its translation is straightforward and depicted in Figure 5.8(d).
5.2. MODEL TRANSLATION

Figure 5.7: Pattern RT01, example applications (cont.)

Pattern CI01

Initialization of a capsule roles’ state diagram is translated with the application of pattern CI01. A state machine is initialized by execution of its initial transition when the capsule role on which it is defined is instantiated. As long as execution of the initial transition path has not yet finished the capsule role is actually stateless and not ready to process the reception of events.

The initial point of every state diagram is translated into a place with the application of pattern SP01. To model initialization of the capsule role a place labeled Start is added, in which a token is placed. Assumption 5.1 justifies the placement of this starting token, which implies instantiation of the entire hierarchy as one atomic action.
Assumption 5.1: Hierarchy instantiation
Because of the lack of run-time code analysis there is no means by which to
tell when and how an optional or plug-in type capsule role is instantiated.
Therefore, all capsule roles are assumed to be fixed. Fixed type capsule
roles are instantiated the moment their parent is instantiated. This means
that all capsule roles are instantiated when the top level capsule role is
instantiated.
For the Océ models the assumptions can be further legalized; the statistics,
see Appendix C, reveal that less than 5% of the capsule role instantiations
are of type plug-in. However, the statistics also show that over 95% of
the instantiations are of type optional. The developers at Océ estimate
that roughly 90% of the optional capsule roles are only made optional to
have some influence on system startup and to be able to provide startup
parameters for a capsule role.

Figure 5.9: Pattern CI01
Capsule role instantiation is depicted in its most simple form in Figure 5.9(a),
its translation in Figure 5.9(b). Places Start and Initial are connected with
transition Start that takes a token from place Start and puts it in place Initial.
Transition Initial is immediately enabled and places the token in place SA.
The transition path from place Start to place SA represents the initialization
of the capsule role.

Pattern BT01
At the end of Section 3.3.6 it is explained that border transitions are transitions
that originate on a composite state. Execution of a border transition starts with
the reception of a signal (trigger) for which no transitions are defined on the
states on the path from the active simple state up to the composite state on
which the border transition is defined.
The mapping \( st : S \to E \times T \) maps a simple state \( s \in S \) onto its outgoing
transitions, \( t \in T \), and the set of triggers \( e \in E \) on which they trigger. Mapping
\( st \) is defined in Formula (5.9), where \( ot(\ldots) \) denotes the outgoing transitions
originating on a certain state.

\[
\forall s \in S : st(s) = \bigcup_{t \in ot(s)} \bigcup_{e \in \text{triggers}(t)} (e, t) \quad (5.9)
\]
The definition of \(st\) is extended as denoted in Formula (5.10), where \(dscds(\ldots)\) denotes the descendants of a state, \(\pi_1(\ldots)\) the first element in a set, \(parent(\ldots)\) the parent of a state, and \(S\) the entire set of states in a model, both simple and composite. Using this extended definition a new mapping \(bt: S \rightarrow E \times T\) can be defined that returns the set of all transitions, including border transitions, for a state and their respective triggers. Mapping \(bt\) is denoted in Formula (5.11).

\[
\forall s \in S: st(s) = \bigcup_{t \in ot(s)} \bigcup_{e \in triggers(t) \land e \in \pi_1(\bigcup_{s_1 \in dscds(s)} st(s_1))} (e, t) \tag{5.10}
\]

\[
\forall s \in S: bt(s) = st(s) \cup bt(parent(s)) \tag{5.11}
\]

The definition of mapping \(bt\) provides a mapping of a simple state to its triggered transitions, i.e. transitions for which the set \(E\) is empty are not taken into account. Therefore, another mapping \(tt: S \rightarrow E \times T\) is introduced. Mapping \(tt\), denoted in Formula (5.12), where \(\epsilon\) denotes the empty set, defines the entire set of transitions for a simple state, including border transitions and transitions for which no triggers are defined. Ignoring transitions that are not triggered from ascendant composite states is validated by Assumption 5.2.

\[
\forall s \in S: tt(s) = bt(s) \cup \bigcup_{t \in ot(s) \land triggers(t) = \emptyset} (\epsilon, t) \tag{5.12}
\]

**Assumption 5.2: Border transition triggers**

The Océ coding guidelines require border transitions to be triggered, i.e. for every border transitions a non-empty set of triggers \(E\) must be defined. Therefore, it is assumed that border transitions are always triggered.

The use of self-transitions as border-transitions might result in the execution of a sequence of exit codes, followed by the transition code, and at last a sequence of entry codes. The translation of entry and exit codes is, in accordance with Assumption 5.3, not taken into account.

**Assumption 5.3: State entry and exit action code**

It is assumed that entry and exit action code blocks are not used in a model, and when used do not contain statements resulting in the sending of signals. The use of entry and exit actions makes it unclear what code is actually being executed upon reception of a certain event. For instance, the Océ coding guidelines explicitly state not to use entry and exit action code blocks. Statistics about RoseRT models that have been generated further justified the assumption as they revealed that in far less than 1% of the state definitions action code, either entry or exit, has been defined.

An example of a border transition defined in a RoseRT model is depicted in Figure 5.10(a). Both state \(SA\) and \(SB\) have no outgoing transitions that trigger on signal \(B\) received on port \(pA\), transition \(v\) can, therefore, be executed from both states. Figure 5.10(b) shows the translation into a Petri net; from place...
CHAPTER 5. ROSE REALTIME MODEL TRANSLATION

(a) RoseRT view

(b) Petri net translation

Figure 5.10: Pattern BT01

SA two transitions can be enabled, v and u, v is a self-transition and thus is the token returned to place SA after execution. Place SB enables v upon reception of signal B. The sets defined by bt(SA) and bt(SB) are \{(pA.A, u), (pA.B, v)\} and \{(pA.B, v)\} respectively.

A somewhat more complicated example showing the application of BT01 is depicted in Figure 5.11; two border transitions are modeled in the RoseRT view. State SA has an outgoing transition that triggers on pA.A, only border transition x is therefore regarded an outgoing transition of SA. The calculated sets for this example are \{(pA.A, u), (pA.B, x)\} for bt(SA) and \{(pA.B, v), (pA.A, w)\} for bt(SB).

(a) RoseRT view

(b) Petri net translation

Figure 5.11: Pattern BT01, extended example

Pattern CS01

Composite state occurrences in RoseRT state diagrams are translated using patterns CS01 and CS02. Every composite state contains an initial point via which it is initialized. When no initial transition is present, non-terminating junction points are used to initialize a composite state. When a transition that originates outside the composite state ends on a non-terminating junction point the transition inside the composite state is executed. This transition ends on any of the points and states in the composite state.
5.2. MODEL TRANSLATION

5.2.1. Pattern CS01

Pattern CS01 is concerned with the translation of composite states that are initialized with a transition that ends on one of its non-terminating junction points. In Figure 5.12(a), transition $u$ is an example of a transition that ends on a non-terminating junction point. Execution of transition $u$ is followed by the execution of $w$ in composite state $SB$. Transition $w$ ends on state $SK$ and thereby initializes the composite state. The translation of this behavior is depicted in Figure 5.12(b). All points and states are translated with the application of pattern SP01, the signal places with pattern SD01.

The translation of a transition $t$, as defined by RT01, is defined to be between two simple states, $SA$ and $SB$ for instance. From this point on a transition can occur between two states, two points or the combination. If it is between two states they might be the same. The transitions in the example are translated with pattern RT01 using these new definitions.

5.2.2. Pattern CS02

CS02 is concerned with the translation of composite states that are initialized by their terminating junction points, i.e. by the incoming transitions ending on one of their terminating junction points. A transition ending on such a junction point triggers the execution of the initial transition, thereby initializing the composite state. An example of such a composite state is depicted in Figure 5.13(a), its translation in Figure 5.13(b).
An additional transition that is enabled by a token in place \(JP1\) is added to transfer the token to the place representing the initial point. This transition models the behavior of the terminating junction point.

Combinations of patterns CS01 and CS02 are also possible, i.e. a composite state might be initialized by its initial point or by incoming transitions.

**Pattern CP01**

Choice points are translated with pattern CP01. In summary, a choice point has two outgoing transitions and one incoming transition, depending on the evaluation of a Boolean expression one of the outgoing transitions is immediately taken. An example of a choice point is depicted in Figure 5.14(a). Although no outgoing signals have been defined for the transitions, it is possible for all three of them to generate these. It is only possible for transitions \(u\) to be triggered.

In the Petri net translation, depicted in Figure 5.14(b), the moment of choice is modeled with the two transitions, \(True\) and \(False\), being enabled by place \(CP1\). The transitions are, again, translated with pattern RT01. Translation of a choice point as described is valid under Assumption 5.4.
5.2. MODEL TRANSLATION

Assumption 5.4: Choice point expression

It is assumed that the Boolean expression on which the decision to execute either the true or false transition is based does not contain statements that send one or more signal(s). The assumption is validated by the Océ coding guidelines and the statistics, listed in Appendix C, that are generated from the test models.

![Diagram (a) RoseRT view and (b) Petri net translation](image)

Figure 5.14: Pattern CP01

Observations regarding patterns CS01 and CP01

After widening the definition of pattern RS01 for the translation of CS01 it might be argued that there is no need for pattern CS01 and CP01. The points and states required for the translation are created with pattern SP01, the transitions with pattern RT01. However, pattern CS01 is added to clearly define the semantic difference with CS02, pattern CP01 because choice points are important modeling constructs and their translation must be completely clear.

Statistics

As for structural patterns, formulas can be defined to calculate lower bounds on the number of places and transitions needed for state machine translations. The calculation of the number of places is performed with Formula 5.13 and Formula 5.14, where \( CR \) denotes the set of capsule roles, \( css(...) \) the composite states contained in a composite state, \( cers(...) \) a capsule roles’ set of encapsulated capsule roles, \( CS \) the set of composite state, \( cps(...) \) the choice points in a composite state, \( sss(...) \) the simple states in a composite state, \( topstate(...) \) denotes the top state of a capsule role, and \( c(...) \) the cardinality of an element. The constant value 2 in Formula 5.14 counts for the start and initial place created for the translation of every composite state.
∀_{cr \in CR} : NrPlaces(cr) = 
NrPlaces(topstate(cr)) \times c(cr) + 
\sum_{ecr \in ecrs(cr)} NrPlaces(ecr) \tag{5.13}
∀_{cs \in CS} : NrPlaces(cs) = 
2 + |cps(cs)| + |sss(cs)| + 
\sum_{ecs \in css(cs)} NrPlaces(ecs) \tag{5.14}

The lower bound on the number of transitions is calculated with Formula (5.15), where \( t_{jps}(\ldots) \) denotes a set of terminating junction points, \( ips(\ldots) \) the set of initial points, \( parent(\ldots) \) the state on which a point is modeled, and \( its(\ldots) \) and \( ots(\ldots) \) the in- and outgoing transitions of a point or simple state. The second line calculates the number of terminating junction points modeled on composite states, they require, as defined in pattern CS01, an additional transition to the initial point. The third line calculates the number of transitions originating on a point for which the number of incoming transitions is greater than zero, it would be a border transition or initial point if otherwise. Triggers can not be specified for transitions originating on points, therefore it is sufficient to count the number of transitions. The fourth line uses the definition of \( tt \), Formula (5.12), to calculate the number of transitions required for the translation of transitions originating on simple states.

∀_{cr \in CR} : NrTransitions(cr) = 
\left[\{p : p \in t_{jps}(cr) \land parent(p) \in CompositeStates_{cr}\}\right] + 
\sum_{p \in points(cr) \land |its(p)| > 0 \lor p \in ip(cr)} |ots(p)| + 
\sum_{s \in sss(cr)} |tt(s)| \times c(cr) + 
\sum_{ecr \in ecrs(cr)} NrTransitions(ecr) \tag{5.15}

5.2.3 Code translation

The set of patterns to translate the structural and behavioral modeling elements is further extended with four code translation patterns. Previous sections already defined translations patterns in which signals were sent from a transition. Sending of signals is always the result of code execution. This section presents the patterns to translate these code fragments.

The analysis performed by the patterns is strictly structural; dynamic behavior is not taken into account and the C++ code itself can, therefore, be abstracted from. The code is analyzed for the presence of send, selection and iteration
5.2. MODEL TRANSLATION

statements. As a code fragment is always a sequence, it can be translated into
an open workflow state machine Petri net [11], which creates all possible code
execution outcomes.

The following elements are present in the abstracted C++ language: a state-
ment, denoted by $C$, the skip action, $\tau$, the signal send action, $a!$, for instance,
statement composition $C.C$, selection statements, $C + C$, and iteration state-
ments, $C^*$. Grammar (5.16) denotes this abstract language.

\[
C \rightarrow \tau \mid a! \mid C.C \mid C + C \mid C^*
\]  
(5.16)

Every element, except for the composition, denoted in the grammar is trans-
lated with a pattern. Initially, a transition is translated as depicted in Fig-
ure 5.15(a). The composition statement is translated by fusing the end and
start place of two translated statements, i.e. the cloud in Figure 5.15(a) is
continuously enlarged by placing translated statements in it. After code trans-
lation the end place of the last statement and the end place of the transition
translation are connected, this is depicted in Figure 5.15(b) in which the cloud
represents the translated statements.

There are a number of less obvious drawbacks to not performing run-time code
analysis resulting from or next to, for instance, not knowing variable values:

1. The structural code analysis extracts functions calls from the code and
parses the corresponding code for send statements. However, it is impos-
sible to extract send statements from functions referenced by function
pointers as these can only be determined at run-time. This means that
for action code blocks in which function pointers are used the structural
code analysis possibly fails to extract every relevant statement.

2. Identifying a function call is not difficult, determining which function is
actually called might be more difficult: the C++ language allows for the
definition of overwritten functions. An overwritten function is a function
with the same name but different parameters. The difference might be in
the number of parameters, in which case it is possible to determine which
variant is being called, or in the type of the parameters, in which case it
is impossible to determine the function being called as type extraction is
not part of the structural analysis.

![Figure 5.15: C++ code translation](image)
Pattern CA01

Pattern CA01 is concerned with the translation of the skip action, the $\tau$ language element. Translation of the skip element is shown in Figure 5.16; the start and end places are connected with a dummy, $\tau$, transition.

![Figure 5.16: CA01 Petri net translation](image)

Pattern CA02

A send action, $\alpha !$, is translated with the application of pattern CA02. Figure 5.17 depicts the translation of a send action, of which examples are shown in Code Listing 5.1. The example code denoted in Code Listing 5.1 should be thought of as being in the action code of one RoseRT transition. The translation of a send action is based on the translation of events as defined in [10].

![Figure 5.17: CA02 Petri net translation](image)

To illustrate the composition and send action consider Code Listing 5.1 which contains two send actions. Assume that the transition for which the code fragment is defined is from state $SA$ to $SB$ and is triggered by signal $A$ received on port $pA$. The four steps in the translation are depicted in Figure 5.18(a) to (d).

Figure 5.18(a) shows the initial translation of the transition, a start and end place are created and RT01 connects the trigger. Code analysis identifies the first send action and connects it to the start place, Figure 5.18(b), the second statement is identified next and composed to the first, Figure 5.18(c), when no more statements are left the end places are connected by means of a $\tau$ transition, Figure 5.18(d).

![Code Listing 5.1: Example of application of composition statement and send action](image)

```java
... 
pA.Z().send();
...
pA.Y().send();
... 
```
5.2. MODEL TRANSLATION

(a) Initial translation

(b) pA.Z().send() translated

(c) pA.Y().send() translated

(d) Translation finished

Figure 5.18: Send and composition statement translation

Pattern CA03

Grammar element $C + C$, the selection statement, is translated with pattern CA03. The Microsoft C++ language reference\(^1\) defines two possible selection statements. These are denoted in Code Listing 5.2 and Code Listing 5.3.

The selection statements can be used recursively, which means that the use of nested selection statements must be anticipated in the analysis process. Code Listing 5.4 denotes an example if selection statement, the Petri net translation of this statement is depicted in Figure 5.19. A default alternative is present in the code, this means that one of the three options is always executed. From the Petri net translation it can be deduced that indeed one of the tree transitions sequence from SelectionStart to SelectionEnd is always chosen.

\(^1\)the RoseRT tool is used in combination with Microsoft Visual Studio 6.0
Code Listing 5.2: Selection statement: *if*

```java
if (expression)
    statement1
[else
  statement2]
```

Code Listing 5.3: Selection statement: *switch*

```java
switch (expression)
    case constant-expression: statement
[default: statement]
```

Code Listing 5.4: Pattern CA03 code view

```java
if (expression) {
    dosomething();
} else if (expression) {
    doanotherthing();
} else {
    dodefaultcase();
}
```

Figure 5.19: Pattern CA03 Petri net translation

The translation depicted in Figure 5.19 shows the translation in which it is assumed that no signal send actions are present in selection expressions. This translation is in accordance with the Océ coding guidelines. However, code investigation revealed that there were many situations in which the coding guidelines were violated. An additional translation of selection statements is defined and used in the case that expressions are to be taken into account. This translation is depicted in Figure 5.20.
5.2. MODEL TRANSLATION

Not all selection statements need to have a default alternative, see Code Listing 5.5, therefore an additional translation of selection statements is introduced. It combines CA01 and CA03, the default case is translated with a $\tau$ transition. The translation is depicted in Figure 5.21. Translation of switch selection statements is not detailed here, as an arbitrary switch statement can be rewritten to an if statement. However, for the translation of switch statements Assumption 5.5 should be taken into account.

**Assumption 5.5: Switch statement analysis**

The Océ coding guidelines disallow for fall-through switch statements. A fall-through statement is a case statement in which code is executed but that is not ended with a break or return statement. Execution of this case is immediately followed by execution of the code in the next case. This behavior is hard to analyze and difficult to trace, therefore it is assumed that all switch statements are well-formed as defined in the guidelines, i.e. each case ends with a break or return statement except when it does not contain any code.

![Figure 5.20: Pattern CA03:1 Petri net translation with expression analysis](image)

**Code Listing 5.5: Pattern CA03 no default alternative**

```java
if (expression) {
    dosomething();
} else if (expression) {
    doanotherthing();
}
```

The translation of selection statements as presented in this section is comparable to the translation of activity diagram decisions as defined in [10].

**Pattern CA04**

Translation of the iteration statement is presented in this section with the introduction of pattern CA04. Actually, two patterns concerning translation of the iteration statement are introduced: CA04.1 and CA04.2. CA04.1 translates an iteration statement into a Petri net in which the number of repetitions is
unbounded, whereas CA04.2 translates it into a net with a limited, by some $x \in \mathbb{N}$, number of repetitions.

The C++ language reference [25] defines three types of iteration statements, which are listed in Code Listing 5.6, Code Listing 5.7, and Code Listing 5.8. These definitions are defined recursively on the element `statement`, which means that selection and iteration statements can be nested an arbitrary number of times.

**Code Listing 5.6: Iteration statement: while**

```cpp
while ( expression )
   statement
```

**Code Listing 5.7: Iteration statement: for**

```cpp
for ( init-expression ; cond-expression ; loop-expression )
   statement
```

**Code Listing 5.8: Iteration statement: do**

```cpp
do
   statement
   while ( expression ) ;
```

**CA04.1**

Pattern CA04.1 translates an iteration statement into a Petri net in which an unlimited number of iterations is possible. An example of a C++ iteration statement, the `while` statement, is denoted in Code Listing 5.9. The
5.2. MODEL TRANSLATION

Figure 5.22: Pattern CA04.1 Petri net translation

unbounded translation is depicted in Figure 5.22, the cloud represents the translation of `dosomething()`. The place between `repetitionEnd` and `repetitionStart` is used to guarantee that it is not possible to execute multiple instances of the loop in parallel.

Code Listing 5.9: Pattern CA04 code view

```java
while (expression) {
    dosomething();
}
```

The translation presented in Figure 5.22 is sufficient for the `while` and `for` iteration statements. For the `do` statement, however, the iteration must be executed at least once. The Petri net translation in which this requirement is satisfied is depicted in Figure 5.23.

The translations depicted in Figure 5.22 and Figure 5.23 both lack the translation of iteration expressions. When expressions are taken into account the translation would have been as depicted in Figure 5.24. However, based on Assumption 5.6 is the translation of expressions in iteration statements omitted.

---

2 only depicted for the `while` and `for` statement
Assumption 5.6: Iteration statement expressions
The Océ coding guidelines strictly prohibit the use of send actions in iteration expressions, as they lead to unpredictable behavior. It would be very hard to give an estimate on the number of messages being sent. Moreover, expressions in iteration statements have very different semantics; for a for statement its expression is only evaluated once, whilst for do or while statements it is executed every iteration.

CA04.2

Pattern CA04.2 is concerned with the translation of an iteration statement into a Petri net that is limited by $x \in \mathbb{N}$ in its number of repetitions. A limited translation is depicted in Figure 5.25. The number of iterations in the example is limited by 100. Upon repetition start 100 tokens are placed in place $p_4$, of which one is removed at the start of every iteration. The cloud represents the translated iteration code. When the expression evaluates to false, modeled by transition $expFails$, a token is placed in place $p_2$ and $p_3$. Transition $returnTokens$ returns all 100 tokens to place $p_4$, which enables transition $cleanUp$. Finally, transition $continue$ is executed and places a token in place $p_7$, which is used to guarantee that no iterations are being executed in parallel.

The defined translation does not correctly translate the semantics of the do
5.2. MODEL TRANSLATION

Figure 5.26: Pattern CA04.2 Petri net translation for a do iteration statement

Figure 5.27: Pattern CA04.2 Petri net translation with expression analysis

iteration statement. Figure 5.26 depicts a translation in which at least one iteration is guaranteed and therefore is a correct translation of the do statement. This translation requires the addition of a transition that models the true evaluation of the iteration expression, \( \text{expHolds} \). For both CA04.1 and CA04.2 a translation for the do iteration statement has been presented, however, under validity of Assumption 5.7 it is chosen not to include these into the translation.

Assumption 5.7: Iteration statement: do

\textit{Code analysis of the Océ models revealed that the do iteration statement is used in less than 1\% of the iteration statements. For this reason is inclusion of the do iteration in the translation regarded future work.}

Although Assumption 5.6 validates the choice not to include iteration expression statements, is a proposal for such a translation presented in Figure 5.27. Future work might be to include this variant of pattern CA04.2 into the translation.

Translation pattern CA04 concludes the discussion of the code analysis patterns.

All translation patterns have been presented. The relation between them is as
follows: the behavioral patterns rely on the structural patterns for the translation of triggers and send events. The code analysis patterns depend on the behavioral patterns as they implement the transitions representing RoseRT transitions.

The translation alternatives considered during research and motivations for not using them are presented in Appendix A.

5.2.4 Transition atomicity

In Chapter 3 it is explained that transition execution in RoseRT follows the run-to-completion paradigm, i.e. transition execution can not be interrupted by the reception of a new event, it is an atomic action. The patterns discussed in the previous sections do not safeguard atomicity.

Transition atomicity is added to the translation patterns with the addition of one global place, P_ATOMICTY, that initially contains one token. When a transition from a certain place, representing a RoseRT simple state, is enabled the token must be in P_ATOMICTY. The atomicity token is returned when a transition terminates on a place representing a RoseRT simple state. As only one atomicity place is created it is safeguarded that no two sequences in the Petri net, representing RoseRT transition sequences between simple states, are executed in parallel.

Figure 5.28(a) depicts patterns RT01 and Figure 5.28(b) CP01 to which the atomicity token has been added. Figure 5.28(b) clearly illustrates how the token is returned only after completion of the entire transition sequence.

5.3 Limitations

The translation presented in the previous sections is not complete. Due to assumptions about the models and restrictions on RoseRT usage are certain model properties and elements not, correctly, translated. These limitations will briefly be explained in this section.

Much of the limitations can be regarded direct consequences of the richness of the RoseRT feature set. RoseRT is a very baroque tool, numerous features
5.3. LIMITATIONS

with their own specific properties and setup options exist. This baroqueness makes it very difficult to define a universally applicable translation.

5.3.1 Static model analysis

The translation of RoseRT models is limited to the translation of design-time elements and properties. Therefore, the semantic differences between the three capsule role types are not taken into account. This limitation is a direct consequence of Assumption 5.1.

Another consequence of only analyzing design-time behavior is the absence of non connected port roles in the translation. This is a direct consequence of pattern SD02 in which it is defined that only connected ports are used to send messages.

5.3.2 Thread analysis

With the introduction of the atomicity place, as explained in Section 5.2.4, the translation actually becomes a translation of a single-threaded RoseRT model. The reason for this is, again, to be found in the analysis being limited to static model properties. It can only be determined at runtime in what thread a certain capsule role will be instantiated. To further emphasize the importance of this limitation is it included as Assumption 5.8.

If multiple threads were to be taken into account it would require an atomicity place for every different thread. The token, initially added to this place, would be shared between all capsule role translations defined to run in the thread.

Assumption 5.8: Single threaded translation

As it can only be specified with the instantiation of a capsule role in which thread it will run, all capsule role instances are assumed to run in the same thread.

5.3.3 Message queue

The RoseRT message queue concept, as explained in Section 3.4, is not translated as such. Before the translation of a capsule role it is not known in which order messages will be received. The queue is defined as a list of received signals placed in a certain order. Assuming only two possible signals to be received on a certain capsule role, namely $A$ and $B$, would already make modeling of the queue infeasible; as can be seen in Figure 5.29.

Figure 5.29 only depicts sequence $A \ast B\ast$, but sequences like $B \ast A\ast$, $B \ast A \ast B \ast A B A\ast$ and all imaginable variants on these are also possible and would all require their own transition sequence. An option to reduce the number of transition sequences would be to abstract over signal types. Abstracting over signal types would only leave one sequence, namely $X\ast$, where $X$ represents the untyped signal. However, it is not possible to abstract over signal types as these are required by the translation of triggers and send action statements.
CHAPTER 5. ROSE REALTIME MODEL TRANSLATION

The message that would in RoseRT be placed at the end of the queue are placed as tokens in places in the Petri net translation. The order in which messages are received is therefore lost in the Petri net. In the Petri net translation the queue therefore modeled as a bag of signals. The bag is defined as the union of all tokens in all places representing in signals.

Modeling the message queue as a bag is actually an overspecification: it is possible to execute a Petri net transition sequence that represents the RoseRT transition sequence that would have been selected in the RoseRT model based on the queue contents. Overspecification also means that is possible to execute numerous transitions paths that could not have been executed in the RoseRT model. Overspecification might, therefore, lead to invalid analysis results.

5.4 Translation of Petri net analysis results back into RoseRT

One of the main goals as set out in Section 1.1 is that it should be possible to translate Petri net analysis results back into RoseRT. The results should be visualized such that the underlying Petri net semantics are hidden from the developer.

Behavior of a RoseRT model is mostly defined in the exchange of signals between different capsule roles. The RoseRT tool uses UML message sequence charts to visualize the exchange of messages between capsule roles. The capsule roles involved in a certain execution path are added to the message sequence chart, arrows between these are drawn to depict the message flow. An example message sequence chart is depicted in Figure 5.30. It shows two capsule roles, client and server, exchanging a sequence of messages. Capsule role client starts by sending message A to the server, which somewhere in the future responds with messages Y and Z. Message B is then send to the server, followed by Y to the client, and finally by again, message A to the server.

A developer can easily trace an execution path from a message sequence chart. From a state machine it follows which transition triggers on which signal and thus which code is executed. The use of message sequence charts in combination with state diagrams makes the identification of a possible error a less laborious task.

The analysis methods as described in this thesis all produce counterexample traces, e.g. for deadlock analysis the counterexample trace is a sequence of transitions leading to a deadlock. To construct the message sequence chart it suffices to extract the transitions in the trace that have output arcs to places representing ports (those created by pattern SD01). When this list of transi-
5.4. TRANSLATION OF PETRI NET ANALYSIS RESULTS BACK INTO ROSERT

![Message sequence chart example](image)

Figure 5.30: Message sequence chart example

For every transition \( t \) in \( CT \) the capsule role on which it is executed can be determined from its name. When all capsule roles that are involved in the generation of the trace have been identified they are added to the message sequence chart. For every outgoing arc of \( t \) that places a token in a port place, a signal between two capsule roles is added to the message sequence chart. The target capsule role can be determined from the name of the place.
This chapter presents means to reduce the number of places and transitions created in a Petri net during a translation. To do so a number of translation optimizations and translation reductions are presented. Optimizations define heuristics on which a decision to not include certain parts of a RoseRT model into a translation is based. Translation reductions are defined as a set of rules that enable the transformation of Petri net place/transition sequences into equivalent sequences requiring less places and transitions. Optimizations are translation options, i.e. dynamic translations are defined with it, reduction rules are applied in a certain translation or not, they are fixed.

The need for reductions and optimizations arose when performing analysis on models provided by Océ-Technologies: the number of places required for a translation is over 100,000, the number of transitions, in some situations, over 1,000,000. This actually changed the goals as set at the beginning of the project. An additional goal is formulated: “to define means by which a Petri net translation of a RoseRT model can be altered such that the number of places and transitions needed for the Petri net is minimal, while keeping semantic equivalence.”

The optimizations are presented in Section 6.1, the reductions in Section 6.2, and, finally, are the results gained from these optimizations and reductions presented in Section 6.4.

6.1 Optimizations

The optimizations to reduce the number of places and transitions used in a translation are presented in this section. The optimizations are mostly based on structural observations. Some of the optimizations presented in this section can not be chosen to be applied in a certain translation: they are implicitly applied in every translation. These are optimizations without which it would be impossible to generate valid Petri nets.

6.1.1 H01: Disregard unconnected ports

Connections for ports unconnected at design time can not be determined by the structural code analysis, therefore, no places representing the signals of such a port need to be created. All RoseRT transitions defined to trigger on a signal received on an unconnected port are translated as if no trigger was defined, i.e. there is no precondition to the execution of this transition. The application of optimization H01 does not affect translation pattern RT01: the only effect is that the set of triggers used by pattern RT01 is limited to those defined on connected ports.

All translations that are based on a model in which unwired ports are modeled contain, due to optimization H01, more execution paths as the original RoseRT model. An illustration is given in Figure 6.1 for which two translations of a
Figure 6.1: Optimization H01

Figure 6.2: Optimization H02

RoseRT model are given. Port \( pB \) is not connected, port \( pA \) is connected. At runtime the model behaves as modeled in Figure 6.1(a), i.e. \( pB.X \) is a precondition to transition \( v \). However, Figure 6.1(b) depicts the actual translation in which \( v \) is enabled as soon as a token is in place \( SB \).

H01 is one of the optimizations that is always applied. Without its application deadlocks might occur since transitions would never be enabled as tokens in the signal places would never arrive. The signals would never arrive as no input arcs are drawn to \( in \) places, and no output arcs are drawn from \( out \) places.

6.1.2 H02: Disregard unconnected capsule roles

An unconnected capsule role is defined as a capsule role for which none of its ports are connected. Unconnected capsule roles can be disregarded in the translation as no signals can be sent to or received from it. Obviously, is the top capsule role the exception for this optimization: it might be unconnected but should always be included. Application of optimization H02 ensures that the Petri net is completely connected, i.e. no detached Petri nets exist.

An example of an unconnected capsule role, \( coffeeBrewer \), is depicted in Figure 6.2. From this view it can be easily deduced that at design time it is not known how communication between \( coffeeBrewer \) and the other capsule roles is intended. Capsule role \( coffeeBrewer \) is, therefore, completely disregarded.
6.1. OPTIMIZATIONS

6.1.3 H03: Relay ports disregarded

The only purpose of relay ports is to forward signals up or down the hierarchy. Signals can only be sent or received at end ports, therefore, it is sufficient to only translate and connect end ports. Application of optimization H03 updates the definition of pattern SD02; only connected, via zero or more relay ports, end ports are translated.

Figure 6.3 depicts two examples of the application of pattern SD01, where the ‘ep’ places denote end ports and the ‘rp’ place a relay port. Figure 6.3(a) denotes the translation without application of optimization H03, Figure 6.3(b) with its application.

6.1.4 H04: Translation of in signals disregarded

One port role can only be connected to exactly one other port role. The application of pattern SD02 creates two places and one transition for every connection. From the Murata [29] reduction rules it follows that this sequence of two places connected by one transition can be reduced to only one place.

Therefore, it is chosen only to translate the out signals of every port. The translation of RoseRT transition triggers as defined in RT01 generated an input arc from the place representing the in signal of a capsule role. Application of H04 removes the places representing in signals, a RoseRT trigger is translated by an arc from the place representing the out signal of the sending capsule role. Figure 6.4 depicts the application of optimization H04. Optimization H04 reduces the number of places involved in port role translations with exactly 50%.
6.1.5 H05: Limited capsule role cardinality

The analysis of different RoseRT models revealed that low level capsule roles are mostly dynamically instantiated. Low level capsule roles are those instantiated at the bottom of the capsule role hierarchy and mainly representing the software implementations of hardware sensors. At design time it is often not exactly known how many instances will be created, this unawareness is translated into a RoseRT model by specifying a theoretical upper bound $X$ for the cardinality of such a capsule role. During translation it results in the capsule role being translated $X$ times.

Before translation a new upper bound $Y$ might be specified, the number of capsule role translations is now defined by $X'$, where $X' = \min(X, Y)$.

6.1.6 H06: Limited port cardinality

An implication of specifying theoretical capsule role cardinality upper bounds is that all ports that might be connected to these instances need a cardinality equal to or greater than this upper bound. They might even be higher since it is possible that connections to multiple instances of different capsule roles are modeled, therefore an optional upper bound limiting the number of port instances may be specified prior to a translation.

The implications of optimizations H05 and H06 are enlisted in Table C.1 in Appendix C. The upper bounds for optimizations H05 and H06 should be carefully chosen and always in consensus with the developers who, best of all, can define realistic upper bounds.

6.1.7 H07: Non-behavioral border transitions not regarded

The border transitions regarded in this section concern border transitions that are self-transitions. The term non-behavioral border transition is introduced to denote border transitions of which the action code does not contain send action statements. The active state does not change with the execution of a border self-transition, which means that translation is not necessary if no send actions statements are defined in its action code. Instead, it is sufficient to generate a dummy transition, $\tau$, that consumes the token from the trigger place.

6.1.8 H08: Introduce trigger place to RT01

The translation of a RoseRT transition $t$ with pattern RT01 is defined as follows (see Section 5.2.2): For each trigger $e \in E$ a transition between $SA$ and $SB$ is created that is enabled by the signal defined in $e$, and which places tokens in the places representing the signals in $S$. The set of triggers is defined by $E$ and the set of generated signals by $S$. An implication of this definition is that for every defined trigger the entire transition, including action code, is translated. Optimization H08 defines a new translation definition for pattern
RT01, explicitly designed for those models in which transitions with multiple triggers often occur.

For every transition $t$ between states $SA$ and $SB$, a place $p_t$ is created indicating that transition $t$ can be fired. For every trigger $e$ in $E$ a transition $t_e$ between $SA$ and $p_t$ is created. From place $p_t$ transition $t$ to $SB$ is added. Transition $t$ should, again, be thought of as a subnet in which the translation of the transition action code is generated.

To clarify this new definition are the optimized and non optimized translations of pattern RT01-MT depicted in Figure 6.5. Figure 6.5(a) depicts the non optimized version of pattern RT01-MT, Figure 6.5(b) the optimized version. Reductions in the number of places and transition may not be immediately apparent, one should imagine application of this optimization in models in which large action code blocks are defined.

6.1.9 H09: Eliminate pattern CA01

The Murata [29] reduction rules, in particular the rule fusion of series places, allow for the elimination of pattern CA01: translation of $\tau$ statements. The non optimized version of pattern CA01 is depicted in Figure 6.6(a), the optimized, and semantically equivalent, version in Figure 6.6(b). Refer to [48] for a correctness proof of this optimization.

6.1.10 H10: Translation of capsule role initialization

The application of pattern CI01 results in a sequence of three places connected by two transitions. There is no need to actually model the $P_{\text{Start}}$ place, only modeling the initial place results in equivalent behavior.
Optimization H10 eliminates the creation of place `Start` in a translation. A translated state machine is initialized by placing a token in place `Initial`.

### 6.1.11 Statistics

In the previous chapter a number of formulas have been defined to calculate lower bounds on the number of places and transitions in a Petri net generated from a RoseRT model. In this section the formulas that need to be redefined in order to show the influence of the optimizations are presented.

Formula 5.3 is redefined to Formula 6.1, where $PR$ denotes the set of connected end ports in a model, $outsignals(...)$ the set of signals that can be sent from a port, $MAXPC$ the maximal port cardinality, and $c(...)$ the cardinality of a port. The definition is influenced by optimizations H01, H03, H04 and H06.

$$\forall pr \in PR: NrPortPlaces(pr) = |outsignals(pr)| \cdot \min(c(pr), MAXPC) \quad (6.1)$$

Formula 5.4 is redefined to Formula 6.2, where $PR$ and $c(...)$ are defined as in Formula 6.1, $CR$ as the set of connected capsule roles, $ecrs(...)$ as a capsule roles’ set of connected encapsulated capsule roles, and $MAXCC$ as the maximal capsule role cardinality. Of influence on the new definition are optimizations H02 and H05.

$$\forall cr \in CR: NrPortPlaces(cr) = \min(c(cr), MAXCC) \cdot \sum_{pr \in PR, cr} NrPortPlaces(pr) + \sum_{ecr \in ecrs(cr)} NrPortPlaces(ecr) \quad (6.2)$$

Formula 5.13 and Formula 5.14 are redefined to Formula 6.3 and Formula 6.4, respectively. In these formulas $CR$ denotes the set of connected capsule roles, $CS$ the set of composite states, $c(...)$ the cardinality of a capsule role, $MAXCC$ the maximal capsule role cardinality, $ecrs(...)$ a capsule roles’ set of connected encapsulated capsule roles. The other functions and constants have not changed. Under optimization H10 has the constant 2 been lowered to 1. Other optimizations of influence are H02 and H05.

$$\forall cr \in CR: NrPlaces(cr) = NrPlaces(topstate(cr)) \times \min(c(cr), MAXCC) + \sum_{ecr \in ecrs(cr)} NrPlaces(ecr) \quad (6.3)$$

$$\forall cs \in CS: NrPlaces(cs) = 1 + |cps(cs)| + |sss(cs)| + \sum_{ecs \in ecss(cs)} NrPlaces(ecs) \quad (6.4)$$
6.2 Reductions prior to Petri net creation

To reduce the number of places and transitions involved in code translations five reduction rules have been defined. These rules are denoted in Reduction Rule (6.5) through Reduction Rule (6.9).

\[
\begin{align*}
    a!b! & \iff (a, b)! \quad (6.5) \\
    a!\tau b! & \iff a!b! \quad (6.6) \\
    a!(b! + c!) & \iff (a!b!) + (a!c!) \quad (6.7) \\
    a!(b!c)!^* & \iff a!(c!b!b!{^*}\tau) \quad (6.8) \\
    a!(b!c)!^* & \tau \iff a! + a!b!(b! + c!){^*}\tau \quad (6.9)
\end{align*}
\]

Application of Reduction Rule (6.5) through Reduction Rule (6.7) is graphically depicted in Figure 6.7. Figure 6.7(a) could be a translation of a random action code block in which two selection statements are executed after each other. The intermediary steps needed to reduce Figure 6.7(a) to Figure 6.7(b) are denoted in (6.10), a visualization of these steps is depicted in Figure D.2 in Appendix D.

![Diagram](image.png)
CHAPTER 6. OPTIMIZATIONS & REDUCTIONS

An application of Reduction Rule (6.8) is depicted in Figure 6.8. Transition $b!$ can be enabled an arbitrary number of times, which is why rule Formula (6.8) can be applied. After application of Reduction Rule (6.5) and Reduction Rule (6.8) the reduced version as depicted in Figure 6.8(b) is obtained. The entire reduction sequence is depicted in Figure D.1 in Appendix D, and denoted in (6.11).

\[
(a_1! + \ldots + a_n!)(b_1! + \ldots + b_m!)
\]

\[
\Leftrightarrow \quad (a_1! + \ldots + a_n!)(b_1! + \ldots + b_m!)
\]

\[
\Leftrightarrow \quad a_1!(b_1! + \ldots + b_m!) + \ldots + a_n!(b_1! + \ldots + b_m!)
\]

\[
\Leftrightarrow \quad (a_1 b_1)! + \ldots + (a_1 b_m)! + \ldots + (a_n b_1)! + \ldots + (a_n b_m)!
\]

Application of Reduction Rule (6.9) is depicted in Figure 6.9. No further reduction sequences are denoted for this rule as it only involves one step.

In Section 5.3 it is explained that the RoseRT message queue is translated as a bag of tokens in different places. Application of the reductions presented in this section is only valid when the message queue is modeled as a bag of tokens, i.e. the message order is irrelevant.

### 6.3 Reductions following Petri net generation

Structural reductions can also be applied following the translation of a RoseRT model into a Petri net.

Part of the masters project of J.M.E.M. van der Werf [48] was to integrate the reductions as defined by Murata [29], Desel and Esparza [7], and Berthelot [6], into the Yasper PNML libraries.
6.4 RESULTS

The thesis investigated the relations between the different reductions. A minimal set of reductions that has to be applied for maximal effect can be deduced from these relations. These are the abstraction, self-loop places, and parallel places rules as defined by Murata and the identity transition and identical transition rules defined by Berthelot.

This minimal set of reductions has been implemented as, an optional, step in the Petri net generation process.

6.4 Results

Measurements on the total amount of places and transitions in a Petri net have been performed both in reduced and non reduced versions of Petri nets created from the coffee machine example and from the real-life model, ModelY, provided by Océ. The results are summarized in Table 6.1. RoseRT statistics about the models can be found in Appendix C. For ModelY one should refer to the statistics for the model limited to an upper bound of 5 and 50 for capsule role and port role cardinalities, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Coffee machine</th>
<th>Océ model, ModelY</th>
</tr>
</thead>
<tbody>
<tr>
<td>not reduced &amp;</td>
<td>283 places</td>
<td>123,216 places</td>
</tr>
<tr>
<td>not optimized</td>
<td>274 transitions</td>
<td>82,480 transitions</td>
</tr>
<tr>
<td></td>
<td>(≈ −0%)</td>
<td>(≈ −85%)</td>
</tr>
<tr>
<td>not reduced &amp;</td>
<td>283 (≈ −0%)</td>
<td>16,891 (≈ −85%)</td>
</tr>
<tr>
<td>optimized</td>
<td>274 (≈ −0%)</td>
<td>18,150 (≈ −78%)</td>
</tr>
<tr>
<td>reduced &amp;</td>
<td>218 (≈ −23%)</td>
<td>6,906 (≈ −94%)</td>
</tr>
<tr>
<td>optimized</td>
<td>207 (≈ −24%)</td>
<td>6,091 (≈ −93%)</td>
</tr>
<tr>
<td>post translation</td>
<td>74 (≈ −74%)</td>
<td>6,100 (≈ −95%)</td>
</tr>
<tr>
<td>reductions applied</td>
<td>64 (≈ −77%)</td>
<td>3,394 (≈ −96%)</td>
</tr>
</tbody>
</table>

Table 6.1: Optimization and reduction results
This chapter discusses the different aspects of the toolkit that has been developed for the analysis of RoseRT models. The first part motivates the design choices, the second discusses the components of the toolkit.

### 7.1 Motivation

#### Development platform

The Microsoft .NET [27] platform has been used for development, with C# [26] as the preferred programming language. The use of .NET as the development platform is mainly motivated by the fact that all the Petri net libraries developed at the Eindhoven University of Technology (TU/e) are also developed on this platform. Developing the toolkits on the same platform allows for a tighter and easier integration.

A comparison with the other programming languages, i.e. Java, C++ and Summit Basic, that have been considered is denoted in Table 7.1. Java and C++ are well known programming languages, Summit Basic is a scripting language provided with the RoseRT tool that enables developers to automate some of their tasks.

It is stated that integration of .NET and C++ libraries with Java is possible via the JNI interface. However, without expensive commercials tools this is a very laborious task and is therefore not considered feasible, which rules Java out as an option.

Summit basic is a scripting language with limited features and is therefore not suited for the implementation of larger applications like the RoseRT analyzer.

The C++ and C# programming languages would both have been equally good options for the implementation of the analysis tools. But, next to the Petri net libraries also being developed in .NET, did further investigation of the external libraries reveal that creating one wrapper for the single RoseRT C++ library is less time consuming then the creation of multiple wrappers for the PNML C# libraries. Moreover, parsing of C++ code with the use of regular expression is much simpler in C# as the engine contains constructs for the balanced matching of brackets for instance.

#### Meta model storage

A meta model is designed and implemented to store extracted RoseRT models in. The meta model itself is presented and discussed in Appendix [2]. The .NET streaming capabilities are applied to physically store a meta model on disk. Using the .NET streaming functionalities imposes restrictions on the implementation of the meta model, but once adhered to, they provide efficient means of saving and loading a model to and from disk.
CHAPTER 7. TOOLKIT

<table>
<thead>
<tr>
<th></th>
<th>.NET</th>
<th>Java</th>
<th>C++</th>
<th>Summit Basic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Oriented (OO) programming</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Integration with .NET libraries</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>-</td>
</tr>
<tr>
<td>Integration with C++ libraries</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>Execution of external tools</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>-</td>
</tr>
<tr>
<td>Regular expression engine</td>
<td>✓</td>
<td>✓/✓</td>
<td>✓/✓</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7.1: Motivations for choosing .NET as development platform

Storing the meta model into a Database Management System (DMBS) has been considered, but as the meta model merely serves as an intermediary format it is not worth the effort to create a database mapping when the .NET streaming capabilities provide a ready-to-use method of storing meta data.

C++ code analysis

The parsing of C++ code blocks on transitions and in operations is performed using regular expressions. Most regular expression engines do not support patterns with balanced constructs, e.g. matching of brackets. When parsing C++ code it is of utmost importance that pairs of brackets can be matched, otherwise it would be unfeasible to detect nested if statements for instance.

The subset of the C++ code fragments regarded useful for the translation into Petri nets does not require an entire C++ parser. The selection and iteration statements can, as an arbitrary send statement, be perfectly matched against a regular expression. Furthermore, it is difficult to construct a parser that operates on only small pieces of code blocks without any context, like for instance variable or operation declarations. It would require the creation of this context every time a code block is parsed.

The .NET regular expressions used to construct a parse tree are presented in Appendix F. The order in which the expressions should be matched against a C++ code block is presented in the next section with the discussion of component CodeAnalysis.

7.2 Components

A toolkit has been implemented that contains the components for the translation of RoseRT models into meta models and Petri nets, for the formal analysis of the Petri nets, and for the translation of analysis results back into the original RoseRT models. The different packages and components in this toolkit are depicted in Figure 7.1. Following will be a discussion of the different packages and components in the toolkit. The executables, i.e. programs, are compo-
ponents of which the name has been printed in bold. The toolkit is implemented such that as much reuse of existing components is applied as possible, reused components are labeled with the stereotype external library.

Package Rational

The Rational package contains the Rose RealTime tool. RRTEI is a component of RoseRT and provided with a RoseRT installation, the library enables developers to write software that communicates with RoseRT. Virtually every action that can be performed within the tool can also be performed via this library.

Package Yasper

Package Yasper contains numerous .NET library files, the ones used in the toolkit are depicted in Figure 7.1. PNMLIO is used to read and write PNML from and to XML files. Actual Petri net instantiations are created with the PNML library, and reductions, as defined in Section 6.3, on these instances are implemented in library PNMLReductions. All these libraries are created and maintained at the Eindhoven University of Technology by the developers of the Yasper tool.

Package Analysis

The Analysis package contains the different analysis tools. Currently, only LoLa is supported as a Petri net analysis tool. LoLa is actually a collection of multiple Petri net analysis methods, two of which have been integrated in the toolkit. Future extensions of the toolkit might integrate more methods or even different tools.

LoLa executable Deadlock is used to verify the absence of deadlocks in a Petri net. The validity of a CTL formula on a given Petri net is verified with executable ModelChecking. Both executables generate counterexample traces when verification fails.

Package Rose RealTime Extractor

The libraries contained in package Rose RealTime Extractor implement the procedures to create a RoseRT meta model. The RRTEIWrapper library is implemented as a wrapper around the RRTEI C++ library. The wrapper makes it possible for .NET programs to communicate with RoseRT. The wrapper is generated with a conversion tool, TlbImp.exe, that is installed with the .NET framework.

Library RRTDataExtractor implements the actual meta model and communicates, via RRTEIWrapper, with RRTEI to create instances of the meta model.
CHAPTER 7. TOOLKIT

The parse and message trees are constructed with the functionalities implemented in the CodeAnalysis library.

Component CodeAnalysis

The component CodeAnalysis contains the procedures to construct the parse and message tree from a certain C++ code block. It does so by iteratively matching the code against the regular expressions as presented in Appendix F.

The process starts with the elimination of comments from a code block. The regular expressions in Code Listing F.1 and Code Listing F.2 match comments, which are replaced by empty strings. The same is done for single and double quoted strings, with the use of the expressions in Code Listing F.3 and Code Listing F.4. On the code that is not eliminated in the previous steps the structural C++ analysis can be performed.

The first part is to identify and adapt non well-formed selection and iteration statements, i.e. a while loop that is not opened and closed with curly braces. The regular expressions matching non well-formed constructs are denoted in Code Listing F.5, Code Listing F.6, and Code Listing F.7. After replacement of the non well-formed constructs the actual parsing can begin. The code is matched against the regular expression in Code Listing F.8 for the extraction of selection and iteration statements. If it matches a switch selection statement the expression in Code Listing F.9 is used to parse the different cases in the switch construct.

Code Listing F.10 is used to parse operation invocations of the format functionX(param1, ...), whilst Code Listing F.11 parses operation invocations of the format var1.functionX(param1, ...).functionY(...). The latter extracts send statements from the code as these are always of the format: port.signal(...).send().

Finally, the parameters of the identified operation invocations are matched against the regular expression in Code Listing F.12. The parameters need to be parsed for the presence of operation invocations, as these might result in the sending of additional signals prior to the invocation of the operation itself.

Other libraries and executables

Two translation libraries are implemented; Trace2RRT and RRT2PNML. The former translates a counter example trace generated by one of the LoLa tools into a RoseRT message sequence chart, the latter translates a meta model instance into a Petri net in PNML format.

RRTDataBrowser is an additional tool that can be used to view all elements in a generated meta model. RoseRTAnalyzer is the executable that actually connects all the different components. It uses RRT2PNML to create Petri nets, package Analysis to analyze these, and Trace2RRT to create counter example message sequence charts of the analysis results.
Figure 7.1: Rose RealTime analysis toolkit components, executables are printed in bold
CHAPTER 7. TOOLKIT

7.3 Approach

The process of translating a RoseRT model into a Petri net, followed by a Petri net analysis, and finally the translation of results back into the RoseRT model is explained in this section. The approach is very similar to the one taken by Gregor Engels et al. in [9] for the model-based verification of properties, which is depicted in Figure 7.2(a).

The approach depicted in Figure 7.2(a) shows the formulation of formal conditions based on the property of a UML model that is to be verified. Our approach, as depicted in Figure 7.2(b), generates a LoLa instance based on the chosen analysis method. This instance is created by setting up compilation parameters that are used in the LoLa source code to compile a specific version of LoLa, optimized for the chosen analysis method.

Both approaches create a translation, a Petri net translation for the approach depicted in Figure 7.2(b) and a graph transformation for the other one.

After specifying what is to be verified and generation of the translation can the model be analyzed. Visualization of the analysis results is where both approaches differ. The approach by Gregor Engels et. al. visualizes the results in a result model, which may use the same modeling formalism as used in the original model. Our approach always visualizes the result in the original UML model.

(a) Steps within model-based validation of properties, taken from [9]  
(b) RoseRT model analysis process

Figure 7.2: Approaches to the analysis of software models

Mapping of components on approach steps

The left part of the approach depicted in Figure 7.2(b) is statically contained in the toolkit and modeled in package Analysis, the different LoLa instances are only compiled once. The RoseRT model is modeled in package Rational, its translation in package Rose RealTime Extractor and library RRT2PNML. The actual analysis is performed from program RoseRTAnalyzer with the invocation of the LoLa instances. The translation of analysis results is performed by library Trace2RRT.

The entire RoseRT analysis process can be explained with the Petri net as depicted in Figure 7.3. From the RoseRT model a metamodel is created with the
7.3. APPROACH

Figure 7.3: Petri net view on RoseRT analysis process

*RRTDataExtractor* component of *RoseRTAnalyzer*. From this metamodel a Petri net is generated by the program with the use of the *RRT2PNML* component. Analysis of this Petri net is performed with a LoLa instance, this analysis evaluates to either true or false. In the false case a counterexample trace is generated that is translated back into the RoseRT model by the *Trace2RRT* component of *RoseRTAnalyzer*. 
Example: coffee machine

The coffee machine example has been referred to numerous times in previous chapters. This chapter provides a detailed description of the model and its translation, as well as a review of the obtained analysis results.

The coffee machine was chosen to serve as a proof-of-concept throughout the research project. The limited scale of the model allowed for a feasible verification of the translation and a comfortable process of tracing analysis results. The use of a small-scale model allowed for an early identification of toolkit design and implementation flaws. Early identification of problems enabled a natural evolution of the implementation such that it could eventually be applied on the real-life models provided by Océ-Technologies B.V..

The first part of this chapter is concerned with the structure and behavior of the coffee machine model, the second part with the discussion of the analysis results.

8.1 Introduction

The coffee machine model comes pre-installed with RoseRT. It models the implementation of a very simple coffee machine. The coffee machine consists of capsule role drivers and the back-end capsule role cuContainer. cuContainer contains four capsule roles that model the internal behavior. Capsule role drivers provides software interfaces to the different hardware parts of the coffee machine. Capsule role cuContainer is modeled as a structural proxy, i.e. it only forwards messages up and down the hierarchy, and has no state machine. The back-end capsule role coffeebrewer serves as a proxy between the other back-end capsule roles. Capsule role sprayer takes care of pouring the coffee into the pot, frontPanel initiates back-end processes as a reaction to user inputs, and plate models the plate on which the pot is placed. A graphical overview of this structure is depicted in Figure B.1 in Appendix B.

Capsule role drivers is used to model the coffee machine process, i.e. the creation and pouring of coffee. The coffee machine as provided does, by default, lack an implementation of this capsule role. However, a number of example processes are provided with the model. One of these, the process of creating exactly one pot of coffee, is chosen to be implemented for analysis purposes. Note, that integration of this example changed the definition of the RoseRT model.

The next sections discuss the different capsule roles in the model. Numerous references to Appendix B will be made during the discussion, it contains images of and details about the transitions in the state diagrams. Before reading the next sections it is advised to first study Section B.1 and Section B.2. Section B.1 presents graphical overviews of the different capsule roles, their structure and their state machines, Section B.2 denotes transitions details.
8.2 Translation

Capsule role *sprayer*

Capsule role *sprayer* controls the spray of hot water over the coffee. Communication with the other back-end capsule roles is realized via port role *coffeeControl*, communication with capsule role *drivers* is realized via port roles *reliefValve* and *boiler*. The different port roles are used to receive or broadcast information about the status of the spraying process, but also to get informed on whether a pot is present or not.

As capsule role *sprayer* is the first to be discussed will its presentation be in greater detail compared to the presentation of the remaining capsule roles. For readability reasons certain figures have, that are also available in Appendix B, been included in this section.

The structure and state diagram of capsule role *sprayer* are depicted in Figure 8.1. The structure diagram shows the internal representation of the ports. In combination with Figure B.1 the structural relationships with other capsule rules can easily be deduced. The state diagram defines the behavior, it is composed of four simple states and eight transitions. The transition details are presented in Table B.1 in Appendix B. The strong relationship with the structure diagram is underlined with the numerous references to the different port roles in the action code blocks.

Capsule role *sprayer* is initialized to state *NotBrewing*. When it is informed that the boiler is no longer empty, the active state becomes *ReadyToBrew*. From there it can be requested to brew a cup of coffee, which is modeled with state *Spraying*. The process of spraying hot water can be interrupted with the reception of a message indicating that there is no pot available. Reception of this message would transfer the active state to *PauseBrewing*, the process is continued when a pot becomes available. From state *Spraying* the active state is returned to *NotBrewing* when it is informed that the boiler is empty.
8.2. TRANSLATION

<table>
<thead>
<tr>
<th>Border transition</th>
<th>NotBrewing</th>
<th>ReadyToBrew</th>
<th>PauseBrewing</th>
<th>Spraying</th>
</tr>
</thead>
<tbody>
<tr>
<td>IgnorePots</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coffeeControl.Pot</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coffeeControl.NoPot</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NotReady</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>coffeeControl.CheckReady</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 8.1: Border transition properties for capsule role sprayer

Border transition *IgnorePots* is used to handle the reception of messages *Pot* and *NoPot* received on port *coffeeControl*, as these are only relevant in states *Spraying* and *PauseBrewing*, respectively. No behavior is defined in transition *IgnorePots*, i.e. no messages are sent. Border transition *NotReady* handles message *CheckReady*, which is received on port *coffeeControl*. The capsule role is only ready in state *ReadyToBrew*, which means that message *NotReady* should be replied in all other states. The explanation of the border transitions is further clarified with Table 8.1, it denotes which border transitions can be executed from which simple state.

Now that the inner workings of capsule role *sprayer* have been explained the translation into Petri nets can be presented. The unreduced and non-optimized translation is depicted in Figure 8.2(a). Translation of border transitions is left out for reasons of simplicity. Figure 8.2(b) depicts the reduced translation, due to the limited size of action code blocks in the coffee machine are the improvements in number of places and transitions relatively small. With reduced version it is meant that all applicable optimizations and the reductions prior to Petri net generation have been applied. Results of the reductions following Petri net generation are not depicted, examples thereof can be found in [48].

The reduced and not reduced translations that do take border transitions into account are depicted in Figure B.5 in Appendix B.

Capsule role *frontpanel*

As explained in the introduction capsule role *frontpanel*, depicted in Figure B.2(d), reacts to user inputs by initiating back-end processes. Its state machine consists of three simple states and four transitions interconnecting them.

The capsule role is idle in states *NotBrewing* and *WaitForStart*. In state *NotBrewing* it can receive a message that a pot of coffee is requested, upon which the active state becomes *WaitForStart*. When the actual brew signal is received the active state is transferred to *Brewing*, when brewing is finished the state machine becomes idle again in state *NotBrewing*. From simple state *WaitForStart* can transition *BrewDenied* be triggered to abort the brew cycle by returning to state *NotBrewing*.

Border transition *IgnoreXtraButtonPush* is modeled to capture buttons being pushed during a brew cycle.

The unreduced and non-optimized translation of capsule role *frontpanel* is depicted in Figure B.6(a). The places to which no transitions are connected
Figure 8.2: Translation of capsule role sprayer
represent port places to which no signals are being sent and on which no transitions are defined to trigger. The reduced and optimized version is depicted in Figure B.6(b). Due to the very limited length of the code blocks the achieved reduction results are limited. The only reduction is obtained by the application of optimization pattern H10.

As capsule role sprayer was used to show the translation of border transitions, capsule role frontpanel will be used to depict the translation of atomic transitions. The translation of capsule role frontpanel including atomic transitions is depicted in Figure B.7(a) and Figure B.7(b), respectively showing the unreduced and reduced translation.

**Capsule role plate**

The three simple states in Figure B.2(e) model the plate of the coffee machine on which the pot is placed during a brew cycle. The plate can only be in state *Warming* when the pot is available and not empty. States *PotEmpty* and *NoPot* model the pot being empty and unavailable, respectively.

The unreduced translation is depicted in Figure B.8a, the reduced translation in Figure B.8b. Translations including border and atomic transitions will be omitted from now on.

**Capsule role coffeeBrewer**

Capsule role coffeeBrewer serves as a structural proxy between the different back-end capsule roles. Its state machine is only modeled by one state, *Forwarder*, that receives and forwards messages from and to the different capsule roles it is connected to. The structure and state machine diagrams for this capsule role are depicted in Figure B.2(b), its translations in Figure B.3(a) (unreduced) and Figure B.3(b) (reduced).

Translation of the if statement on transition *BrewRequest* is shown in the lower right of the Petri net, the number of places and transitions involved can be greatly reduced with the application of reduction rule Formula (6.10), as it is shown in the lower right of the reduced translation.

**Capsule role drivers**

Capsule role drivers models the front-end of the coffee machine, it consists of a sequence of nine states connected by eight transitions. The sequence of states models the brew cycle of exactly one pot of coffee, it starts at state *Waiting* and finishes at state *Done*. The capsule role diagrams are depicted in Figure B.2(a), the unreduced and reduced translations in respectively Figure B.9(a) and Figure B.9(b).
CHAPTER 8. EXAMPLE: COFFEE MACHINE

8.3 Analysis

Coffee machine analysis results are presented in this section. The coffee machine has been analyzed for the absence of deadlocks, as well as for the presence of a well-defined end state.

The message sequence charts depicted in this section are automatically generated in the RoseRT tool, the tool determines the layout of the chart and is not very good at it. The sequence numbers mostly indicate the correct order, whilst the ordering from top to bottom often seems to be mixed up.

Absence of deadlocks

Deadlock analysis revealed the presence of a possible deadlock in the system. A sequence of 27 transitions forms the counterexample trace, of which 21 transitions initiate the sending of a signal. Translation of this trace into a RoseRT message sequence chart is depicted in Figure 8.3.

All five capsule roles are involved in the generation of a deadlock. Analysis of the trace revealed it as actually being a deadlock trace, but an expected one. The path leads to a system deadlock state in which the active state of capsule role drivers is Done. It is correct that no further transitions can be taken from there, however, as this use case only considers the implementation of exactly one brew cycle, i.e. it is implemented to end in state Done, should this not be regarded an actual design flaw.
8.3. ANALYSIS

Figure 8.4: MC2 trace generated from coffee machine analysis

Presence of an end state

The example is implemented such that capsule role drivers is initialized to simple state Waiting and, after one brew cycle, ends up in simple state Done. Using analysis method AMC2 with Waiting as the start state and Done as the end state generated a counterexample trace. After a short inspection it revealed that the trace was generated as a result of not being able to model the message queue in the Petri net.

Deliberately introducing an error in the coffee machine lead to sensible analysis results and an interesting counterexample trace. To generate the counterexample is statement t.boiler.boilerNotEmpty().send(); on transition Prepare of capsule role drivers commented.

The message sequence chart representing the trace is depicted in Figure 8.4. Analysis of the trace revealed that the error originated at the selection statement in the action code of transition BrewRequest on state forwarder of capsule role coffeeBrewer. Procedure AreDevicesReady() returns false in the generated trace because capsule role sprayer send signal NotReady. Capsule role sprayer returns not ready as it never received the boilerNotEmpty signal and therefore is still in state NotBrewing. The false evaluation of procedure AreDevicesReady() leads to the sending of signal BrewDenied to capsule role frontPanel.

Transition BrewDenied from simple state WaitForStart is triggered by this signal. Its execution does not generate any other events. Capsule role drivers is never informed about the brew being denied and the end state will not be reached. The system terminates in an undefined end state.

The same counterexample trace can also be obtained by an AD1, deadlock, analysis but is presented here to exemplify the application of a model checking (AMC) analysis.
9 Océ model analysis

This chapter presents the results of the analyzes that have been performed on different real-life RoseRT models provided by Océ-Technologies B.V.. Statistics about ModelX and ModelY presented in this chapter can be found in Appendix C.

9.1 Example: ModelX

The ultimate goal of the research project was to verify certain properties of the real-life printer model, ModelX\(^1\), developed at Océ. The model chosen for analysis contains all the elements that are actually available on the printer into account, i.e. everything between hardware sensors at the lowest level and user inputs at the highest level.

Translation process

Generation of the meta model took quite some time for this model, around three hours. Profiling the analyzer program revealed the Component Object Model (COM) layer between the RRTEI library and the actual RoseRT program being the major reason for this lack of performance.

Translation of the meta model into PNML format never finished (at least not within 92 hours). Profiling the application led to the observation that the PNML libraries gradually slowed down the process as the number of places went over 30 thousand. It seemed that the underlying hash table generated too many collisions at this point, playing with its parameters did not increase performance. It seems that the PNML libraries do not scale very well to Petri nets of this size. This observation did not come as a surprise to the developers as the libraries were initially developed for use with the Yasper tool in combination with small-scale Petri nets.

Analysis process

Without a generated Petri net no analysis tasks can be performed. The problems observed in the generation of the Petri net emphasize the need for solid reduction and optimization techniques, such as those presented in Section 6.1 and Section 6.2.

9.2 Example: ModelY

After the failure to translate and analyze Océ RoseRT model ModelX it was tried to analyze ModelY\(^2\). ModelY is about one-third of the size of ModelX.

\(^1\)Fictional name used
\(^2\)Fictional name used
It models a scanner as present in copiers, i.e. a part of a copier. The model contains sufficient elements to define interesting analysis tasks.

**Translation process**

The meta model for *ModelY* was generated in about one hour. This was, again, due to the overhead of the COM layer.

The translation of the meta model into PNML format finished, with optimizations being applied, in three hours. Statistics on the number of places and transitions in the generated Petri net are denoted in Table 6.1 in Section 6.4.

**Analysis process**

After generation of the Petri net it was, naively, tried to perform a deadlock, AD1, analysis. The LoLa tool ran for 12 hours without generating counter examples. Investigation of the model learned that many busy-waiting processes are modeled in the system. Obviously, the presence of busy-waiting loops prohibits the presence of deadlocks in the system.

A number of instances were defined for analysis tasks AMC1, AMC2, and AMC3. The verification of the CTL formulas on a Petri net of this size made LoLa run out of memory in less than 10 seconds. The analysis was conducted on a 2.4GHz processor with 2 Gigabytes of RAM. Experimenting with the LoLa setup parameters and reductions techniques did not bring any solution.

There was a small difference in the evaluation of analysis task AMC1 and analysis tasks AMC2 and AMC3: the latter two ran out of memory even faster. This is most probably due to the sub formula contained in these tasks.

**9.3 Example: ModelZ**

*ModelZ* is the stubbed version of a single component, available in both *ModelX* and *ModelY*. The stubs are implemented as use cases. Translation of this model into a Petri net is only a matter of minutes.

The stubs are modeled as use cases which makes an AD1 analysis irrelevant as the system is modeled such that it ends in a deadlock state when all use cases have been executed.

One AMC1 instance has been defined, its verification generated a counter example trace. Inspection revealed that it was actually the lack of runtime code analysis that generated the trace. The violation of the formula was caused by a sequence of *if* statements being taken that could not have been executed in a real environment.

The component allows for the definition of an AMC2 instance, that concerns the initialization process of the component. Execution of this AMC2 instance

---

Fictional name used
evaluated to True, it can be concluded, within translation limitations, that initialization of this component is correctly modeled.

Another AMC2 instance evaluated to False for the same reason as the AMC1 analysis task evaluated to False. Some other checks all evaluated to True for the following reason: the stubs are implemented as use cases. This means that signals that are expected to be returned are hard coded, which makes verification of certain paths very trivial as they only rely on the reception of signals generated in the stubs.

The stubbed model did not allow for the definition of an AMC3 analysis.
Recommendations and conclusions

This Master Thesis concludes with the presentation of conclusions and recommendations. The conclusions elaborate on the applicability of the developed theories and tools. Recommendations might concern the addition of extra features to the toolkit, or the improvement of currently implemented features, but also hints and suggestions on how to proceed with the Petri net based verification of RoseRT models.

10.1 Conclusions

The main goal, “to define a translation of UML-RT models into Petri nets that allow for the identification of errors in the original Rose RealTime model”, has been partly achieved. A translation of RoseRT models into classic Petri nets has been, albeit with known limitations, defined. However, Petri nets generated from larger RoseRT models can not be verified for certain correctness properties with the tools currently available. The tools, or better the analysis techniques implemented in these, are correct, but physical limitations, e.g. the availability of computer memory, makes it that the tools do not scale up to Petri nets counting tens of thousands of places.

Most of the limitations known to exist in the translation are a direct consequence of the baroqueness of the RoseRT tool. To overcome some of the difficulties resulting from this baroqueness a number of assumptions are formulated. When eliminating a certain feature from the rich element set available in RoseRT by means of an assumption it is very important to ensure that intended system behavior is not limited in the generated Petri net. The Petri net translation of a RoseRT model, therefore, almost always is an over specification of the intended behavior.

Together with some RoseRT users four analysis tasks are formulated. These tasks represent some of the most common questions developers would like to have answered about their software models. The analysis tasks return a counterexample Petri net trace when verification of a property fails. Translation of a trace into a message sequence chart in the original model hides developers from the underlying Petri net semantics. The use of UML message sequence charts for this purpose turned out to be very effective. A developer can easily see the sequence of messages being exchanged between the different capsule roles that lead to a, possible, error state in the system.

Performing analysis tasks on the larger models led to the formulation of a fourth subgoal. This subgoal concerned the reduction of the number of places and transitions in a generated Petri net. A number of optimization patterns and reduction rules are formulated and implemented for the achievement of this subgoal. Carefully applying these patterns and rules makes it possible to reduce the size of Petri nets with up to 95 percent.
10.2 Recommendations

1. **Code analysis** Only structural code analysis is implemented in the current toolkit. Verification revealed the need for more in-depth code analysis as counter example traces were generated that could not have been executed during run-time. These traces were almost always the result of a wrongly presumed True of False evaluation of a selection statement. Implementing a more in-depth code analysis might introduce questions about the applicability of classic Petri nets, it could very well turn out that for code evaluations reasons it is better to use Colored Petri nets for instance.

2. **Thread analysis** The current translation assumes RoseRT models being single-threaded. This is hardly ever the case for real-life systems. All kinds of processes run in different threads. In which thread a process runs is determined at run-time. The translation should be updated such that an atomicity token is added for every thread. Using a formalism different from classic Petri nets might even allow for the definition of thread priorities.

3. **Message queue** In classic Petri nets the RoseRT message queue can not be modeled in any way feasible. However, the message queue determines the behavior of a system for a large part. Future version of the toolkit should therefore implement means to model the message queue. It might, again, very well be that a translation into another Petri net formalism is required to do so.

4. **Petri net storage format** The EPNML format is currently used for the storage of generated Petri nets. The Yasper PNML libraries, however, do not scale very well to large Petri nets. Either, it should be tried to further improve performance of the libraries, or to find another storage format.

5. **Petri net analysis tools** Verification of different models showed that the model checking instance of analysis tool LoLa is not suited for large Petri nets. It should be investigated whether for model checking it would not be better to use specified model checkers, like $\mu$CRL or SPIN.

6. **Additional analysis tasks** Four analysis tasks are incorporated into the toolkit. The toolkit is developed in such a way that it allows for an easy addition of analysis tasks. Conformity or controllability analysis could for instance be other interesting analysis methods.

7. **Usage guidelines** Some of the limitations and recommendations presented in this thesis are directly derivable from the abundance of options that are implemented in RoseRT. As semantics of these features are not very thoroughly defined is it difficult to formulate a translation that is applicable in every situation. Further research might include an evaluation on which options should or should not be incorporated in a translation. Results from this study should preferably be formulated in the form of usage guidelines as those presented in Section 5.4.
Translation Alternatives

A.1 Patterns CP01 and CS01

In patterns CP01 and CS01, see Section 5.2.2, it is defined that every RoseRT transition between two places or points is translated separately. An alternative is to precalculate the possible transition paths that can be executed upon reception of a certain trigger. Precalculation is possible as only transitions originating at a state can be triggered.

Consider the translation of a choice point as defined by pattern CP01 in Figure A.1(b), the alternative translation is depicted in Figure A.1(c). In the latter is the choice point eliminated from the translation and replaced by two transitions representing the sequences $t$ followed by $True$ and $t$ followed by $False$.

The example depicted in Figure A.3 shows a more complex example of the application of patterns CP01 and CS01. The alternative translation, Figure A.3(b), effectively visualizes how the alternative greatly reduces the number of places and transitions involved in the structural translation.

Despite the, seemingly, great reduction in number of places and transitions needed for the translation are four reasons presented why not to use this alternative.

1. The reduction is only apparent when C++ code translation is not taken into account. Suppose that transition $t$ in Figure A.3 contains a 1000 send action statements. The transitions from $S1$ to $S2$, $S3$ and $S4$, in Figure A.3(b), would all be refined with the translation of these 1000 send action statements, leading to at least 1000 additional places for every transition. The translation as depicted in Figure A.3(b) would only require the translation of the 1000 send action statements once.

2. The model semantics as they are modeled in the original RoseRT model are completely lost. All sequential moments of choice are translated into one single moment of choice. Whilst in the original model it could be the case that the outcome of the expression in the second choicepoint can only be determined after evaluation of the expression in the first, does the alternative translation move this point to the transitions following place $S1$.

3. Translation of Petri net analysis results into the RoseRT model becomes more difficult as it is not immediately visible which statement was executed on what transition. Furthermore, when multiple paths from one state to another exist it might be undecidable, without additional means, to determine which path was chosen.

4. In a RoseRT model it is not difficult to model a loop between two choice points, as depicted in Figure A.2. Existence of a loop does make precalculation unfeasible as indefinitely many paths from $S1$ to $S2$ exist.
APPENDIX A. TRANSLATION ALTERNATIVES

A capsule role instance is translated the number of times as specified by its cardinality or by the upper bound inflicted by optimization H05. Obviously are the number of places and transitions needed for its translation also multiplied by this cardinality. An alternative would be to translate the instance only once and change the definition of pattern CI01. The cardinality of an instance would specify the number of tokens to be placed in the starting place of the translation.

Figure A.4(a) depicts a, simplified, translation of two capsule role instances; a client with cardinality two and a server with cardinality one. The server is able to send a message A to both clients, it is safeguarded that indeed both instances receive the signal. The alternative is depicted in Figure A.4(b). The number of places and transitions needed is much lower but is not safeguarded that both ‘instances’ process one reception of a signal on port pA. It might as well be that the first case consumes both tokens and returns to the initial place. As safeguarding correct reception handling is considered of greater importance than the reduction in number of places and transitions, is the alternative not implemented.
A.2. CAPSULE ROLE INSTANCES

(a) RoseRT view

(b) Petri net alternative translation

(c) Petri net translation

Figure A.3: Alternative translation to patterns CP01 and CS01

(a) Petri net translation

(b) Petri net alternative translation

Figure A.4: Alternative capsule role instance translation
Example: coffee machine details

B.1 RoseRT model

Figure B.1: RoseRT Coffee machine model hierarchy
APPENDIX B. EXAMPLE: COFFEE MACHINE DETAILS

(a) Capsule role drivers details

Figure B.2: RoseRT Coffee machine model
B.1. ROSERT MODEL

Figure B.2: RoseRT Coffee machine model
### B.2 Transition details

#### Capsule Role: coffeeBrewer

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Simple state: Forwarder**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrewComplete</td>
<td>sprayer.BrewingComplete</td>
<td>frontpanel.BrewingComplete().send();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrewRequest</td>
<td>frontpanel.BrewRequested</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoPot</td>
<td>warmer.NoPot</td>
<td>sprayOff();</td>
</tr>
<tr>
<td>Pot</td>
<td>boiler.boilerOn</td>
<td>sprayOn();</td>
</tr>
</tbody>
</table>

**Capsule Role: sprayer**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Simple state: NotBrewing**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoilerNotEmpty</td>
<td>boiler.boilerNotEmpty</td>
<td>-</td>
</tr>
</tbody>
</table>

**Simple state: ReadyToBrew**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ready</td>
<td>coffeeControl.CheckReady</td>
<td>coffeeControl.Ready().send();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brew</td>
<td>coffeeControl.Brew</td>
<td>boiler.boilerOn().send();</td>
</tr>
</tbody>
</table>

**Simple state: Spraying**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoilerEmpty</td>
<td>boiler.BoilerEmpty</td>
<td>boiler.boilerOff().send();</td>
</tr>
</tbody>
</table>

**Simple state: PauseBrewing**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoPot</td>
<td>coffeeControl.NoPot</td>
<td>coffeeControl.BrewingComplete().send();</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pot</td>
<td>coffeeControl.Pot</td>
<td>coffeeControl.BrewRequested().send();</td>
</tr>
</tbody>
</table>

**Capsule Role: frontpanel**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Simple state: NotBrewing**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrewRequested</td>
<td>coffeeControl.BrewButtonPushed</td>
<td>coffeeControl.BrewRequested().send();</td>
</tr>
</tbody>
</table>

**Simple state: WaitForStart**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrewDenied</td>
<td>coffeeControl.BrewDenied</td>
<td>indicatorLight.indicatorOff().send();</td>
</tr>
</tbody>
</table>

**Simple state: Brewing**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BrewComplete</td>
<td>coffeeControl.BrewingComplete</td>
<td>indicatorLight.indicatorOn().send();</td>
</tr>
</tbody>
</table>
B.2. TRANSITION DETAILS

### Border transitions

<table>
<thead>
<tr>
<th>Transition Role</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IgnoreXraButtonPush</td>
<td>brewButton.brewButtonPushed</td>
<td>brewButton.brewButtonNotPushed</td>
</tr>
</tbody>
</table>

### Capsule Role: plate

<table>
<thead>
<tr>
<th>Simple state: PotEmpty</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: PotNotEmpty</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PotEmpty</td>
<td>warmer.potEmpty</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PotNotEmpty</td>
<td>warmer.potNotEmpty</td>
<td>warmer.warmerOn() send()</td>
<td></td>
</tr>
</tbody>
</table>

### Simple state: Warming

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoPot</td>
<td>warmer.warmerEmpty</td>
<td>warmer.warmerOff().send();</td>
</tr>
<tr>
<td>PotNotEmpty</td>
<td>warmer.potNotEmpty</td>
<td>warmer.warmerOff().send();</td>
</tr>
</tbody>
</table>

### Simple state: NotReady

<table>
<thead>
<tr>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoPot</td>
<td>warmer.warmerEmpty</td>
<td>warmer.warmerOff().send();</td>
</tr>
<tr>
<td>PotNotEmpty</td>
<td>warmer.potNotEmpty</td>
<td>warmer.warmerOff().send();</td>
</tr>
</tbody>
</table>

### Capsule Role: drivers

<table>
<thead>
<tr>
<th>Simple state: Waiting</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare</td>
<td>start.Start</td>
<td>t_warmer.potEmpty().send(); t_boiler.boilerNotEmpty().send(); t_brewButton.brewButtonPushed().send();</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: Light</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IndicatorOff</td>
<td>t_indicatorLight.indicatorOff</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: Check</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HeatBoiler</td>
<td>t_boiler.boilerOn t_reliefValve.valveClose</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: BoilerValveOn</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>WaterStartToFill</td>
<td>t_boiler.boilerOn t_reliefValve.valveClose</td>
<td>t_warmer.potNotEmpty().send();</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: BoilerOff</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TurnOnLight</td>
<td>t_indicatorLight.indicatorOn t_boiler.boilerOff t_reliefValve.valveOpen</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: ValveOff</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boiler</td>
<td>t_indicatorLight.indicatorOn t_boiler.boilerOff t_reliefValve.valveOpen</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simple state: Valve</th>
<th>Transition</th>
<th>Trigger(s)</th>
<th>Action code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valve</td>
<td>t_indicatorLight.indicatorOn t_boiler.boilerOff t_reliefValve.valveOpen</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table B.1:** Coffee machine transitions details for back-end capsule roles
B.3 Petri net translations

The images in this section, starting on the next page, depict the translations of the capsule roles as given in the previous sections. Every image shows both a reduced and unreduced version of the translation. The translation of capsule role \textit{coffeeBrewer} is shown in Figure B.3, the translation of \textit{sprayer} in Figure B.4 and Figure B.5, where the latter depicts the translation including border transitions.

Figure B.6 and Figure B.7 depict the translation of capsule role \textit{frontpanel}, where Figure B.7 shows the translation with atomic transitions. Figure B.8 of capsule role \textit{plate} and, finally, is the translation of capsule role \textit{drivers} depicted in Figure B.9.
B.3. PETRI NET TRANSLATIONS

Figure B.3: Translation of capsule role coffeeBrewer
Figure B.3: Translation of capsule role coffeeBrewer

Reduced
Figure B.4: Translation of capsule role sprayer

(a) Unreduced
Figure B.4: Translation of capsule role sprayer

(b) Reduced
Figure B.5: Translation of capsule role sprayer with border transitions
Figure B.5: Translation of capsule role sprayer with border transitions
B.3. PETRI NET TRANSLATIONS

Figure B.6: Translation of capsule role frontpanel
Figure B.7: Translation of capsule role frontpanel with atomic transitions
Figure B.8: Translation of capsule role plate

(a) Unreduced

(b) Reduced
Figure B.9: Translation of capsule role *drivers*
Figure B.9: Translation of capsule role *drivers*
### Static model properties

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Coffee machine</th>
<th>ModelY¹</th>
<th>ModelY²</th>
<th>ModelX¹</th>
<th>ModelX²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Protocols</td>
<td>20</td>
<td>95</td>
<td>95</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>In signals</td>
<td>95</td>
<td>495</td>
<td>495</td>
<td>658</td>
<td>658</td>
</tr>
<tr>
<td>Out signals</td>
<td>86</td>
<td>563</td>
<td>563</td>
<td>808</td>
<td>808</td>
</tr>
<tr>
<td>Max. number of in signals</td>
<td>12</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Min. number of in signals</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Max. number of out signals</td>
<td>7</td>
<td>13</td>
<td>13</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Min. number of out signals</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classes</th>
<th>Coffee machine</th>
<th>ModelY¹</th>
<th>ModelY²</th>
<th>ModelX¹</th>
<th>ModelX²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>20</td>
<td>441</td>
<td>441</td>
<td>573</td>
<td>573</td>
</tr>
<tr>
<td>Attributes</td>
<td>4</td>
<td>720</td>
<td>720</td>
<td>1,087</td>
<td>1,087</td>
</tr>
<tr>
<td>Operations</td>
<td>0</td>
<td>1,080</td>
<td>1,080</td>
<td>1,497</td>
<td>1,497</td>
</tr>
<tr>
<td>Defining behavior¹</td>
<td>0</td>
<td>0.00%</td>
<td>107</td>
<td>9.91%</td>
<td>107</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Capsules</th>
<th>Coffee machine</th>
<th>ModelY¹</th>
<th>ModelY²</th>
<th>ModelX¹</th>
<th>ModelX²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capsules</td>
<td>26</td>
<td>138</td>
<td>138</td>
<td>231</td>
<td>231</td>
</tr>
<tr>
<td>Attributes</td>
<td>6</td>
<td>550</td>
<td>550</td>
<td>892</td>
<td>892</td>
</tr>
<tr>
<td>Operations</td>
<td>7</td>
<td>459</td>
<td>459</td>
<td>817</td>
<td>817</td>
</tr>
<tr>
<td>Defining behavior²</td>
<td>4</td>
<td>57.14%</td>
<td>141</td>
<td>30.72%</td>
<td>141</td>
</tr>
<tr>
<td>Ports</td>
<td>195</td>
<td>715</td>
<td>715</td>
<td>1,387</td>
<td>1,387</td>
</tr>
<tr>
<td>Non-public ports</td>
<td>78</td>
<td>40.00%</td>
<td>341</td>
<td>47.69%</td>
<td>341</td>
</tr>
<tr>
<td>Cardinality &gt; one</td>
<td>0</td>
<td>0.00%</td>
<td>102</td>
<td>14.27%</td>
<td>102</td>
</tr>
<tr>
<td>Timer ports</td>
<td>12</td>
<td>6.15%</td>
<td>85</td>
<td>11.89%</td>
<td>85</td>
</tr>
<tr>
<td>Frame ports</td>
<td>3</td>
<td>1.54%</td>
<td>59</td>
<td>8.25%</td>
<td>59</td>
</tr>
<tr>
<td>Unwired ports</td>
<td>18</td>
<td>9.23%</td>
<td>250</td>
<td>34.97%</td>
<td>250</td>
</tr>
<tr>
<td>Minus timer &amp; frame</td>
<td>3</td>
<td>1.54%</td>
<td>106</td>
<td>14.83%</td>
<td>106</td>
</tr>
<tr>
<td></td>
<td>Coffee machine</td>
<td>ModelY</td>
<td>ModelX</td>
<td>ModelX</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>--------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>Relay ports</td>
<td>87</td>
<td>44.60%</td>
<td>74</td>
<td>145</td>
<td></td>
</tr>
<tr>
<td>End ports</td>
<td>108</td>
<td>55.38%</td>
<td>64</td>
<td>1.242</td>
<td></td>
</tr>
<tr>
<td>Notification ports</td>
<td>0</td>
<td>0.00%</td>
<td>71</td>
<td>146</td>
<td></td>
</tr>
<tr>
<td>Hierarchy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capsule roles</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cardinality &gt; one</td>
<td>0</td>
<td>0.00%</td>
<td>499</td>
<td>1.190</td>
<td></td>
</tr>
<tr>
<td>Fixed cr’s</td>
<td>0</td>
<td>100.00%</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Optional cr’s</td>
<td>0</td>
<td>0.00%</td>
<td>574</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Cardinality &gt; one</td>
<td>0</td>
<td>0.00%</td>
<td>182</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Port count</td>
<td>31</td>
<td>2.428</td>
<td>1.023</td>
<td>6.489</td>
<td></td>
</tr>
<tr>
<td>CR instances</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Capsule role instances</td>
<td>7</td>
<td>49.097</td>
<td>633</td>
<td>118.739</td>
<td>1.962</td>
</tr>
<tr>
<td>Fixed cri’s</td>
<td>7</td>
<td>100.00%</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Optional cri’s</td>
<td>0</td>
<td>0.00%</td>
<td>49.683</td>
<td>602</td>
<td></td>
</tr>
<tr>
<td>Plug-in cri’s</td>
<td>0</td>
<td>0.00%</td>
<td>222</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Port role instances</td>
<td>31</td>
<td>639.104</td>
<td>10.009</td>
<td>1.521.439</td>
<td>31.561</td>
</tr>
<tr>
<td>Timing pri’s</td>
<td>2</td>
<td>6.45%</td>
<td>25.306</td>
<td>376</td>
<td>60.211</td>
</tr>
<tr>
<td>Frame pri’s</td>
<td>0</td>
<td>0.00%</td>
<td>162</td>
<td>63</td>
<td>423</td>
</tr>
<tr>
<td>Unwired pri’s</td>
<td>2</td>
<td>6.45%</td>
<td>126.218</td>
<td>2.139</td>
<td>300.535</td>
</tr>
<tr>
<td>Unwired timing pri’s</td>
<td>2</td>
<td>6.45%</td>
<td>25.306</td>
<td>376</td>
<td>60.211</td>
</tr>
<tr>
<td>Unwired frame pri’s</td>
<td>0</td>
<td>0.00%</td>
<td>162</td>
<td>63</td>
<td>423</td>
</tr>
<tr>
<td>State machines</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>States</td>
<td>28</td>
<td>249.847</td>
<td>3.365</td>
<td>594.892</td>
<td>10.655</td>
</tr>
<tr>
<td>Composite states</td>
<td>7 25.00%</td>
<td>75.074 30.05%</td>
<td>1.060 31.50%</td>
<td>178.743 30.05%</td>
<td>3.322 31.18%</td>
</tr>
<tr>
<td>------------------</td>
<td>----------</td>
<td>--------------</td>
<td>-------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Simple states</td>
<td>21 75.00%</td>
<td>174.733 60.95%</td>
<td>2.305 68.50%</td>
<td>416.149 69.95%</td>
<td>7.335 68.82%</td>
</tr>
<tr>
<td>Entry codes</td>
<td>1 3.57%</td>
<td>4 &lt; 0.00%</td>
<td>4 0.12%</td>
<td>28 &lt; 0.00%</td>
<td>22 0.21%</td>
</tr>
<tr>
<td>Defining behavior</td>
<td>1 3.57%</td>
<td>4 &lt; 0.00%</td>
<td>4 0.12%</td>
<td>8 &lt; 0.00%</td>
<td>8 0.08%</td>
</tr>
<tr>
<td>Exit codes</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>1 &lt; 0.00%</td>
<td>1 0.01%</td>
</tr>
<tr>
<td>Defining behavior</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
</tr>
<tr>
<td>Junction points</td>
<td>78 1.274.751</td>
<td>16.937</td>
<td>3.036.904</td>
<td>16.937</td>
<td>54.279</td>
</tr>
<tr>
<td>Terminating jp’s</td>
<td>74 94.87%</td>
<td>924.878 72.55%</td>
<td>12.291 72.67%</td>
<td>2.203.964 72.57%</td>
<td>73.535 72.83%</td>
</tr>
<tr>
<td>Jp’s on cs</td>
<td>8 10.26%</td>
<td>649.250 50.93%</td>
<td>8.371 49.42%</td>
<td>1.545.462 50.89%</td>
<td>26.550 48.91%</td>
</tr>
<tr>
<td>Jp’s on ss</td>
<td>70 89.74%</td>
<td>625.5001 49.07%</td>
<td>8.566 50.58%</td>
<td>1.491.442 49.11%</td>
<td>27.729 51.09%</td>
</tr>
<tr>
<td>Choice points</td>
<td>0 50.650</td>
<td>881</td>
<td>121.180</td>
<td>2.885</td>
<td></td>
</tr>
<tr>
<td>Initial points</td>
<td>7 75.074</td>
<td>1.060</td>
<td>178.743</td>
<td>3.322</td>
<td></td>
</tr>
</tbody>
</table>

### Code Analysis

<table>
<thead>
<tr>
<th>Code Analysis</th>
<th>Coffee machine</th>
<th>ModelY</th>
<th>ModelY</th>
<th>ModelX</th>
<th>ModelX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitions</td>
<td>42 751.512</td>
<td>42 10.598</td>
<td>42 1.791.426</td>
<td>42 34.079</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>31 251.818</td>
<td>31 3.690</td>
<td>31 601.282</td>
<td>31 12.240</td>
<td></td>
</tr>
<tr>
<td>Border transitions</td>
<td>16 4.811.991</td>
<td>16 85.317</td>
<td>16 1.145.867</td>
<td>16 275.780</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>5 1.839.945</td>
<td>5 22.076</td>
<td>5 4.387.605</td>
<td>5 77.118</td>
<td></td>
</tr>
<tr>
<td>Transitions from choice points</td>
<td>0 101.300</td>
<td>0 1.762</td>
<td>0 242.360</td>
<td>0 5.770</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>0 429</td>
<td>0 135</td>
<td>0 1.488</td>
<td>0 0.08%</td>
<td></td>
</tr>
<tr>
<td>Initial transitions</td>
<td>6 49.902 6.64%</td>
<td>6 628 5.93%</td>
<td>6 118.729 6.63%</td>
<td>6 1.952 5.73%</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>1 49.717</td>
<td>1 542</td>
<td>1 118.249</td>
<td>1 1.699 4.99%</td>
<td></td>
</tr>
<tr>
<td>Transitions from jp on cs</td>
<td>4 349.873</td>
<td>4 4.446</td>
<td>4 832.940</td>
<td>4 14.746</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>2 174.514</td>
<td>2 2.144</td>
<td>2 415.144</td>
<td>2 6.796</td>
<td></td>
</tr>
<tr>
<td>Transitions from jp on ss</td>
<td>32 250.437</td>
<td>32 3.562</td>
<td>32 597.397</td>
<td>32 11.611</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>22 26.168</td>
<td>22 654</td>
<td>22 63.175</td>
<td>22 2.380</td>
<td></td>
</tr>
<tr>
<td>Self-transitions on ss</td>
<td>9 75.628</td>
<td>9 123.154</td>
<td>9 181.080</td>
<td>9 4.135</td>
<td></td>
</tr>
<tr>
<td>Defining behavior</td>
<td>6 990</td>
<td>6 215</td>
<td>6 3.226</td>
<td>6 0.18%</td>
<td></td>
</tr>
<tr>
<td>Initial transitions</td>
<td>7 75.074</td>
<td>7 1.060</td>
<td>7 178.743</td>
<td>7 3.322</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX C. MODEL STATISTICS

<table>
<thead>
<tr>
<th>Code</th>
<th>Coffee machine</th>
<th>ModelY</th>
<th>ModelX</th>
<th>ModelX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send action statements</td>
<td>43</td>
<td>2,031,155</td>
<td>26,158</td>
<td>4,866,604</td>
</tr>
<tr>
<td>In expressions</td>
<td>2 4.05%</td>
<td>38 &lt; 0.00%</td>
<td>36 0.15%</td>
<td>78 &lt; 0.00%</td>
</tr>
<tr>
<td>Selection statements</td>
<td>1</td>
<td>747,645</td>
<td>17,501</td>
<td>1,811,442</td>
</tr>
<tr>
<td>if statements</td>
<td>1 100.00%</td>
<td>747,418</td>
<td>99.97%</td>
<td>1,810,500</td>
</tr>
<tr>
<td>switch statements</td>
<td>0 0.00%</td>
<td>227 0.03%</td>
<td>226 1.29%</td>
<td>942 0.05%</td>
</tr>
<tr>
<td>Iteration statements</td>
<td>0</td>
<td>4,427</td>
<td>2,662</td>
<td>9,552</td>
</tr>
<tr>
<td>do statements</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
<td>0 0.00%</td>
</tr>
<tr>
<td>while statements</td>
<td>0 0.00%</td>
<td>2,862</td>
<td>64.65%</td>
<td>1,186</td>
</tr>
<tr>
<td>for statements</td>
<td>0 0.00%</td>
<td>1,565</td>
<td>35.35%</td>
<td>1,466</td>
</tr>
</tbody>
</table>

Table C.1: Generated statistics for the coffee machine, ModelX, and ModelY

1. Capsule and port cardinality are unlimited
2. Capsule role cardinality is limited to 5, port role cardinality to 50
3. With defining behavior it is meant that signals are being sent in the code.
4. Composite state count includes the top state of every state machine.
5. jp = junction point, cs = composite state
Reduction rules examples

Figure D.1: Application of reduction rules in Formula (6.5), and Formula (6.8)
APPENDIX D. REDUCTION RULES EXAMPLES

Figure D.2: Application of reduction rules in Formula (6.5) through Formula (6.7)
.NET metamodel

The meta model that is used as an intermediary format between the RoseRT model and the Petri net is presented in this appendix. The meta model can roughly be divided in four distinct sections; one part that stores the structural elements of a RoseRT model, another part that stores the behavioral elements, a part in which a C++ parse tree is stored, and at last a part in which a message tree is stored.

The entire meta model is implemented in C# class files and can therefore be easily presented with UML class diagrams. Figure E.1 depicts the meta model that stores the structural elements of a RoseRT model. The main classes in this diagram are Capsule and CapsuleRole. Capsule and Class both specialize class Classifier, which is composed of Operations and Attributes. Operations have both a parse tree and a message tree.

Next to the Operations and Attributes is a capsule also composed of Ports. As explained in Chapter 5 is the structure hierarchy constructed from capsule roles, modeled by the class CapsuleRole. In the meta model the concept of capsule role instances is introduced with the class CapsuleRoleInstance. An instance is composed of CapsuleRole and PortRole instances. Class PortRole is an abstract class, i.e. it is either a RelayPortRole or an EndPortRole.

The meta model for the behavioral RoseRT elements is depicted in Figure E.2. The most generic representation of a point or state in a RoseRT model is given by the Vertex class. It is specialized by classes PointVertex and StateVertex. The different points, choice, junction and initial, in a RoseRT model are specializations of the PointVertex class. Class CompositeState is a specialization of the StateVertex class. A CompositeState instance is composed of PointVertex specializations and of other CompositeState instances. It represents both the composite and the simple state in a RoseRT model. A composite state that is not composed of other composite states is referred to as a simple state.

The class Transition represents the transitions in a RoseRT model. The attributes ParseTree and MessageTree are required for the analysis of the action code defined on the transition. An arbitrary number of Trigger instances might be defined for a Transition. Both PointVertex and CompositeState are composed of Transition instances. The composition relation defines the outgoing transitions of a PointVertex and the border transitions for a CompositeState.

The meta models in Figure E.3 and Figure E.4 both represent trees used to identify the sending of signals in some C++ code block. The latter is a transformation of the former and is used in the Petri net generation process. The former is used to create a parse tree from the C++ code blocks, the tree is made out of Node instances.

A node is either a SelectionNode, an IterationNode, or a CodeNode. The five specializations of a CodeNode are ConditionNode, ParameterNode, OperationNode, SelectionAlternativeNode, and InvocationNode. Class ConditionNode is used to represent conditions in for instance a selection or iteration statement, a ParameterNode represents parameters in an operation invocation. Class OperationNode is composed of InvocationNode instances.
For instance, code fragment `port.signal().send()` results in the instantiation of an `OperationNode` that is composed from three `InvocationNode` instances, the first for fragment `port`, the second for `signal`, and the third for `send`.

A `MessageTree` instance is a transformation of a `ParseTree` instance into a tree that is only concerned with statements that send signals. The `InvocationNode` instances are, if applicable, translated into `SendActionLeaf` instances. The message tree is created in the same order as the parse tree which means that the sequence of the statements in the C++ code is adhered to. Due to this sequencing is it possible to transform `ConditionNode` and `ParameterNode` instances into `SendActionLeaf` instances.

An instantiated message tree is even further permuted when the reduction rules, as presented in Section 6.2, are applied. Note, that statement ordering might be lost during this permutation process.
Figure E.1: UML class diagram for the static elements of a RoseRT model
Figure E.2: UML class diagram for the behavioral elements of a RoseRT model.
Figure E.3: Parse tree UML class diagram

Figure E.4: Message tree UML class diagram
Regular expressions

Single line comment

Code Listing F.1: Regular expression: single line comment

//[^\n]*

Block comment

Code Listing F.2: Regular expression: block comment

/[*]
(?!(?![\*]/)
  (?:(?:.\[\n\])?)*
)*\[*]/

Single quoted string

Code Listing F.3: Regular expression: single quoted string

' (?:(?:\\')|.\n)*?'

Double quoted string

Code Listing F.4: Regular expression: double quoted string

" (?:(?:\\")|.\n)*"
Non well-formed else and do statements

**Code Listing F.5: Regular expression: non well-formed else and do statements**

```regex
(else|do){1}
\[
  (?:\n
  (\([^\()]*
   (?: (?<Open>\(\)
        (\([^\()]*
         ((?<Close-Open>\))
        )*
      )+
    )+
  )*
  (?(Open)(?!))
)
\)\[
\[
(\[^\();\]*)[;]
```

Non well-formed for statement

**Code Listing F.6: Regular expression: non well-formed for statement**

```regex
(for){1}
\[
  (?:\n
  (\([^\()]*
   (?: (?<Open>\(\)
        (\([^\()]*
         ((?<Close-Open>\))
        )*
      )+
    )+
  )*
  (?(Open)(?!))
)
\)\[
\[
(\[^\();\]*)[;]
```
Non well-formed if, else if, while, and switch statements

```
Code Listing F.7: Regular expression: non well-formed if, else if, while, and switch statements

(else|if|else if|while|switch){1}
{ls}*
(?:
 \(\n\)
 (?:
  (?<Open>\()
  [^\(\)]*
  (?:
   (?<Close-Open>\))
  [^\(\)]*
  )*  
  (?<Close-Open>)
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+  
  (?<Open>\))
  [^\(\)]*
  )+)  
\)
{ls}*
(\{[^\}]*\})\[;]
```

Selection and iteration statements

Code Listing F.8: Regular expression: Selection and iteration statements

```antlr
(else\|if|else|if|while|for|do|switch){1}
[\s]*
(?::
  {[^>\s]*
   (?::
     (?::
       (?<Open>\()
       [^>\s]*
       (?<Close-Open>\))
     [^>\s]*
   )*
   (?<Open>(?!)\))
  )
  [^\s]*
  {[^{\s]*
   (?::
     (?::
       (?<Open>{)
       [^\s]*
       (?<Close-Open}>))
     [^\s]*
   )*
   (?<Open>(?!)\))
  )
  [\s]*
  {[^\s]*
   (?::
     (?::
       (?<Open>\{)
       [^\s]*
       (?<Close-Open}\))
     [^\s]*
   )*
   (?<Open>(?!)\))
  )
  [\s]*
  {[^\s]*
   (?::
     (?::
       (?<Open>\{)
       [^\s]*
       (?<Close-Open}\))
     [^\s]*
   )*
   (?<Open>(?!)\))
  )
  [\s]*
```
Switch statement case analysis

Code Listing F.9: Regular expression: switch statement case analysis

```
(code|default)\{1\}\{(?::|\n)*?\}?\{\}.\n)*?

(?>
 \\{ (?<-LEVEL>)
 | \\} (?<-LEVEL>)
 | 
?\( | \) (. | \n)
)+?
?\(LEVEL)\)?
}
(?=break|return)(((?:\.|\n)*?)(?:;){1}
```

Direct operation invocation

Code Listing F.10: Regular expression: direct operation invocation

```
( (?<!\{?:(?:::|\[.\]|->)\s*)\s*([a-zA-Z0-9\[\]_\-]+)\s*)\s*([a-zA-Z0-9\[\]_\-]+)\s*)
(?:break|return)(((?:\.|\n)*?)(?:;){1}
```
Non-direct operation invocation

Code Listing F.11: Regular expression: non-direct operation invocation

```
\s*
([a-zA-Z0-9\[\]_\-]+)\s*
(?:
  \s*
  \s*\[(?:\([^\(\)]*\)|[^\(\)]*)*\]\s*\]
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
  
  \s*
  \s*\[
  \s*\)
```
Operation invocation parameters

Code Listing F.12: Regular expression: Operation invocation parameters

```
(\s*([a-zA-Z0-9\[\]_\-]+)\s*)(?:\(.*\))?
```

```
Bibliography


state diagram, 19, 39
structure diagram, 13

transition, 19, 23, 39
  border, see border transition, 12
  initial, 20, 41
  path, 23, 29, 41
  selection rules, 21
  self, see self-transition

translation pattern
  behavioural, 38
  structural, 34

trigger, 19, 20, 23, 39, 66

UML-RT, 1
use case view, 9

view
  component, see component view
  deployment, see deployment view
  logical, see logical view
  use case, see use case view