EINDHOVEN UNIVERSITY OF TECHNOLOGY

# From napkin sketches to reliable software

*Citation for published version (APA):*
Engelen, L. J. P. (2012). *From napkin sketches to reliable software*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven.
https://doi.org/10.6100/IR740040

*DOI:*
10.6100/IR740040

*Document status and date:*
Published: 01/01/2012

*Document Version:*
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

*Please check the document version of this publication:*

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

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

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

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

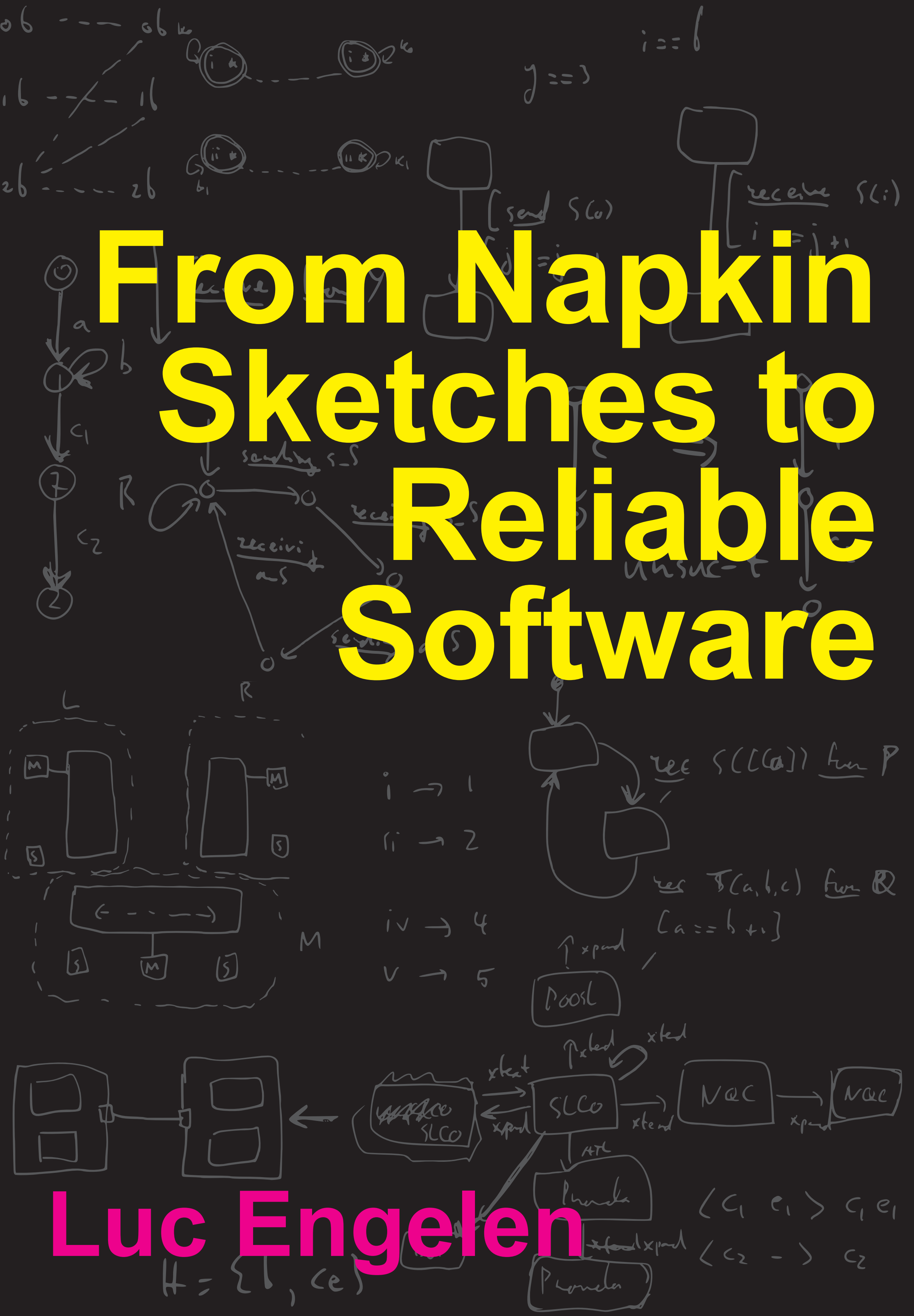
www.tue.nl/taverne

*Take down policy*
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 05. Oct. 2023


From Napkin Sketches to Reliable Software
Luc Engelen

# From Napkin Sketches to Reliable Software

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duin, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op dinsdag 11 december 2012 om 16.00 uur

door

Lucas Johannes Petrus Engelen

geboren te Nijmegen

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. M.G.J. van den Brand

Copromotor:
dr. S. Andova

**From Napkin Sketches to Reliable Software**

Luc Engelen

Promotor: prof.dr. M.G.J. van den Brand
(Eindhoven University of Technology)

Copromotor: dr. S. Andova
(Eindhoven University of Technology)

Additional members of the core committee:

prof.dr.ir. T. Basten (Eindhoven University of Technology)
prof.dr. P.D. Mosses (Swansea University)
prof.dr.ir. A. Rensink (University of Twente)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2012-11.

Part of the work in this thesis has been carried out as part of the Ideals project with ASML as the industrial partner, the Falcon project with Vanderlande Industries as the industrial partner, and the KWR 09124 project LithoSysSL. The Ideals project and the Falcon project fall under the responsibility of the Embedded Systems Institute. The Ideals project is partially supported by the Netherlands Ministry of Economic Affairs under the SenterNovem TS (TSIT3003) program, and the Falcon project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3290-2

© L.J.P. Engelen, 2012.

Printed by the print service of the Eindhoven University of Technology

*All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronically, mechanically, photocopying, recording or otherwise, without prior permission of the author.*

# Acknowledgements

Life as a PhD student has various advantages. For example, as part of my job, I got to visit Portugal, England, Belgium, Germany, South-Korea, and France, and if my planning had been a little better, I could have visited the Czech Republic as well. For this and other reasons, I would like to thank my promotor Mark van den Brand for hiring me and giving me the opportunity to experience all the benefits of a career as a PhD student. Additionally, I thank Mark for his guidance and support over the years.

Around the start of the final year of my project, Suzana Andova agreed to be my copromotor. From that moment on, the rate of my scientific output in terms of papers increased significantly, for which I would like to thank Suzana. Without her confidence and experience as guidance, finishing this thesis would have taken much more time.

Besides Mark and Suzana, Marcel van Amstel and Anton Wijs were co-authors of some of the papers that form the basis of this thesis. I enjoyed the discussions we had concerning the research we performed, and I learned a lot from both cooperations.

Furthermore, I would like to thank the reading committee, consisting of Twan Basten from the Eindhoven University of Technology, Peter Mosses from Swansea University, and Arend Rensink from the University of Twente, for reviewing this thesis.

During most of my employment at the Software Engineering and Technology group, I shared an office with Marcel van Amstel, Jeroen Arnoldus, and Zvezdan Protić. Later on, we relocated and were joined by Yanja Dajsuren, Arjan van der Meer, Ulyana Tikhonova, and Bogdan Vasilescu. I enjoyed the time we spent together in our office, during summer schools, at workshops, and at the Efteling, and I would like to thank all of them for that. Furthermore, I thank all my other colleagues and former colleagues at the Software Engineering and Technology group for providing a pleasant working environment, and Tom Verhoeff in particular for noticing an oversight with significant consequences in one of the model transformations developed by Marcel and me.

It took some time to finish this thesis after my PhD project ended, and I thank Harold Weffers for allowing me to work on it during my employment at LaQuSo.

Finally, I would like to thank my family, friends, volleyball team mates, and my girlfriend Sandra for their support and the much appreciated distractions they provided.

*Luc Engelen*
Eindhoven, August 2012

# Table of Contents

**Acknowledgements** **i**

**Table of Contents** **iii**

**1 Introduction** **1**
1.1 Problem Statement . . . 3
1.2 Research Questions . . . 5
1.3 Outline and Origin of Chapters . . . 6
1.4 Suggested Method of Reading . . . 8

**2 Integrating Textual and Graphical Modeling Languages** **9**
2.1 Introduction . . . 9
2.2 UML Activities and Surface Languages . . . 10
2.3 Specification of the Surface Language . . . 11
2.4 Grammarware . . . 14
2.5 Modelware . . . 16
2.6 Other Applications of our Approach . . . 18
2.7 Case Study . . . 18
2.8 Related Work . . . 19
2.9 Conclusions and Future Work . . . 21

**3 Simple Language of Communicating Objects** **23**
3.1 Metamodel . . . 23
3.2 Concrete Syntax . . . 26
3.3 Target Languages . . . 28
3.4 Semantic Gaps and Platform Gaps . . . 29
3.5 Model Transformations . . . 30
3.6 Sequences of Transformations . . . 43
3.7 Simplified SLCO . . . 45
3.8 Implementation . . . 46

**4 Exploring the Boundaries of Model Verification** **49**
4.1 Introduction . . . 49
4.2 Approach . . . 51
4.3 Comparison of Transformations . . . 52
4.4 Experiments . . . 54
4.5 Discussion . . . 59
4.6 Related Work . . . 59
4.7 Conclusions and Future Work . . . 59

**5 Prototyping the Semantics of a Domain-Specific Modeling Language** **61**
5.1 Introduction . . . 61
5.2 Prototyping Semantics . . . 63
5.3 Visualization . . . 69
5.4 Verification . . . 71
5.5 Related Work . . . 72
5.6 Conclusions and Future Work . . . 72

**6 Reusability and Correctness of Endogenous Model Transformations** **75**
6.1 Introduction . . . 75
6.2 Model Transformations for SLCO . . . 77
6.3 Correctness of Model Transformations . . . 82
6.4 Related work . . . 88
6.5 Conclusions and Future Work . . . 89

**7 Evolution of a Domain-Specific Modeling Language** **91**
7.1 Introduction . . . 91
7.2 Development Process . . . 92
7.3 Evolution . . . 93
7.4 Related Work . . . 99
7.5 Conclusions and Future Work . . . 101

**8 Checking Property Preservation of Refining Transformations** **103**
8.1 Introduction . . . 103
8.2 Background . . . 105
8.3 LTS Transformations . . . 109
8.4 Checking Property Preservation . . . 112
8.5 Experimental Results . . . 118
8.6 Related Work . . . 119
8.7 Conclusions and Future Work . . . 120

**9 Conclusions** **123**
9.1 Contributions . . . 123
9.2 Future Work . . . 126

**Bibliography** **129**

**A Software Tools** **139**
A.1 ASF+SDF and the Meta-Environment . . . . . . . . . . . . . . . . . . . . . 139
A.2 openArchitectureWare . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141
A.3 ATL Transformation Language . . . . . . . . . . . . . . . . . . . . . . . . 142
A.4 Dot and Graphviz . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 142

**B Operational Semantics of SLCO** **143**
B.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
B.2 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
B.3 State Machines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 146
B.4 Initialization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150

**C Two Transformations for SLCO** **153**
C.1 Simple Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
C.2 General Transformation . . . . . . . . . . . . . . . . . . . . . . . . . . . 155

**D Correctness of a Transformation** **159**

**E Case Studies Concerning Property Preservation** **165**
E.1 ACS, 1394-fin, and Wafer Stepper . . . . . . . . . . . . . . . . . . . . . . 165
E.2 Broadcast . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166
E.3 Alternating Bit Protocol . . . . . . . . . . . . . . . . . . . . . . . . . . . 167

**Summary** **169**

**Curriculum Vitae** **171**

**IPA Dissertation Series** **173**

# Chapter 1

# Introduction

In model-driven software engineering (MDSE), models play an important part in the software engineering process [103]. MDSE combines domain-specific modeling languages (DSMLs) [32] with model transformations and is aimed at automatically transforming models written in DSMLs to various artifacts, such as source code and formal models for verification and simulation. DSMLs are modeling languages that offer, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. They allow developers to create models using constructs that describe the problem domain, instead of the solution domain. In traditional software engineering, developers manually create implementations based on designs, whereas in MDSE, model transformations are employed to transform models to implementations automatically. Essentially, the models that are transformed to implementations form the designs of software systems, and automated transformation is meant to prevent errors in the implementations that result from misinterpretations of these designs.

The work described in this thesis was initiated within the Ideals project [19]. This project focused on improving the evolvability of software-intensive high-tech systems by developing methods, techniques, and tools. A lack of proper abstractions for the components and subsystems of a complex embedded system was identified as one of two major causes of the large effort required to maintain and develop these systems. Because of this lack of proper abstractions, little correspondence was perceived between design documents, written on a high level of abstraction, and their implementations, implemented using constructs on a low level of abstraction. To tackle this problem, the application of MDSE in combination with DSMLs was investigated.

ASML, the world's leading manufacturer of lithography systems for the semiconductor industry, participated in the Ideals project as the industrial partner. At the time of the project, the Unified Modeling Language (UML) [86] was gradually introduced as a modeling language for software within ASML, and experiments with MDSE were performed. The UML is named the de facto modeling language for software in industry and offers a number of separate views on systems, in the form of graphical diagrams. One of the experiments with MDSE entailed the investigation of deriving formal models for

performance analysis from UML models. To perform the analysis, the Parallel Object-Oriented Specification Language (POOSL) [108] was used. The benefits of this formal language had become clear because previous experiments had shown that POOSL correctly predicted important properties of a newly developed component at an early state of the development process. However, to spare ASML's software engineers from learning the syntax and semantics of POOSL, or any other formal modeling language, models for these languages should be derived from UML models. In that way, engineers could still benefit from the results of formal analysis, without having to learn multiple modeling languages. In short, UML models should be the starting point for both implementations and various artifacts for analysis.

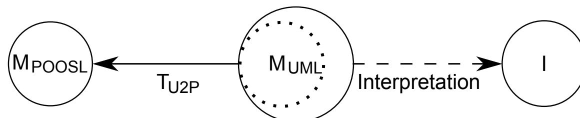



Figure 1.1: Envisioned outcome of the Ideals project

Figure 1.1 schematically depicts the desired outcome of the part of the Ideals project that concerned MDSE. It shows that information is distilled from a large UML model $M_{UML}$ and transformed to a POOSL model $M_{POOSL}$ by means of a model transformation $T_{U2P}$. The dotted circle illustrates that only a part of the information contained in the UML model is used to create the POOSL model. The primary purpose of model $M_{UML}$ is to serve as the design of an implementation $I$ of a system. Software engineers interpret model $M_{UML}$ and develop implementation $I$ manually, as depicted by the dashed arrow.

While developing the transformation from UML to POOSL and experimenting with the creation of suitable UML models, a number of problems surfaced. The diagrams of the UML are not necessarily complete or consistent [73, 74], and its lack of formal semantics gives rise to different interpretations of the constructs it offers. The positive side of this is the freedom it offers its users, which is one possible explanation for the popularity of the language [72]. Although UML models are allowed to be inconsistent and incomplete, however, models that are the starting point for a transformation to POOSL need to be consistent and complete. This requirement gives rise to very elaborate UML models, and creating such models using commercially available graphical editors proved to be cumbersome. Furthermore, the constructs of the UML were not adequate for describing models suited for performance analysis. The language does not offer the appropriate abstractions for all aspects of the desired performance model, and some of the constructs offered by POOSL had no equivalent counterpart in the UML.

When the Ideals project was finished, we decided to create a new DSML to replace the UML as a starting point for the transformation to POOSL because of the aforementioned difficulties. Additionally, this newly developed DSML was intended to be a suitable starting point for transformations to other formalisms and to have an intuitive graphical syntax similar to that of the UML. We named this DSML the Simple Language of Communicating Objects (SLCO). Inspired by the Falcon project [7], we decided to use a small system of interoperating conveyor belts as a case study for SLCO. The overall challenge of the Falcon project was developing a fully integrated and automated logistics warehouse of the future. Conveyor belts form an important part of such a warehouse. Another goal for our DSML thus became to automatically generate software for the controllers of such conveyor belts, based on high-level descriptions specified in the DSML. For verification of SLCO models by means of model checking [26], we decided to develop

a transformation to Promela, the specification language of the model checker Spin [55].

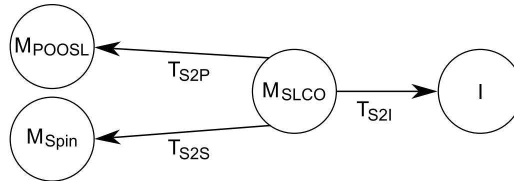



Figure 1.2: A single SLCO model for code generation and formal analysis

Figure 1.2 schematically depicts the automated generation of a model $M_{POOSL}$ for performance analysis, a model $M_{Spin}$ for verification, and an implementation $I$ for a system of interoperating conveyor belts from a model $M_{SLCO}$ specified using the newly developed DSML. The models for formal analysis and the implementation are generated by means of the model transformations $T_{S2P}$, $T_{S2S}$, and $T_{S2I}$.

The development of SLCO and its accompanying model transformations was also triggered by the desire to investigate the internal and external quality of model transformations. The quality of the definition of a model transformation is referred to as the internal quality of a transformation, and the quality of the process of transforming a source model to a target model is referred to as the external quality of a model transformation [3]. The research performed by Van Amstel mainly focuses on the internal quality of model transformations [3], whereas the work described in this thesis is concerned with the external quality of model transformations and their correctness in particular.

To successfully develop the new DSML and the accompanying model transformations, a number of problems needed to be solved.

## 1.1 Problem Statement

There are a number of prerequisites for automatically generating reliable software from high-level descriptions. One of the prerequisites is that the models that form these descriptions should provide an unambiguous specification of the systems they describe. The need for unambiguous models can only be satisfied if the DSML used to specify these models has a formal syntax and semantics.

Additionally, to successfully describe systems on a high level of abstraction using a DSML, this language must offer all the appropriate constructs for the domain at hand. This means that the desired behavior and structure of the resulting software must be expressible in the DSML that is used to describe this software.

Furthermore, to automatically generate software implemented using constructs on a low level of abstraction from designs using concepts on a high level of abstraction, refining model transformations are needed that add implementation details to the models that form these designs. Instead of interpreting a model and manually constructing the corresponding implementation, a software engineer can then generate an implementation from a model by applying one or more model transformations. Implementation details are added to the model in each transformation step, until the resulting model contains sufficient details for deployment on an execution platform.

Because models and model transformations are the primary artifacts in MDSE, being able to validate them during all phases of their development is another important prerequisite. One way of validating models entails using existing languages and tools. By

transforming DSML models to models supported by existing tools, these tools can be used for the validation of (parts of) the DSML models. For instance, model $M_{POOSL}$ shown in Figure 1.2 can be used to simulate model $M_{SLCO}$, and $M_{Spin}$ can be used to simulate or verify $M_{SLCO}$. However, the usefulness of this transformational approach to validation is directly related to the correspondence between the DSML and the languages used for validation. Only those constructs of a DSML that have a counterpart in the languages used for validation can be translated, and only the translatable parts of a DSML model can be validated using the tools for these languages. If not all constructs of a DSML have counterparts in existing languages, custom tools, for instance for the generation and visualization of state spaces, are needed that are specifically built for the validation of DSML models.

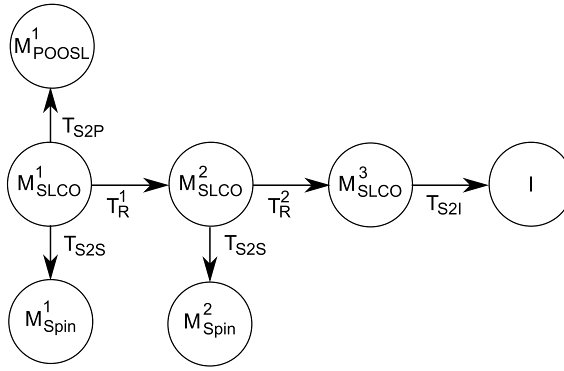



Figure 1.3: Generating an implementation in small steps

Because we add implementation details to models using refining model transformations only, these transformations form a significant part of the design of the resulting software. When generating software by applying model transformations to an initial model, this model and the applied model transformations together define the resulting software in its entirety. Early validation of model transformations is therefore equally important as early validation of models. The validation of model transformations is meant to assess whether these transformations correctly refine models. For each source model and a set of properties that hold for this model, a refining transformation is considered to be correct if the properties also hold for the target model. In this way, an implementation of a system satisfying certain properties can be generated by refining an abstract model that forms the design of this system, if this model satisfies the given properties.

To simplify the development and validation of model transformations, we decompose the transformations into small steps. Splitting a monolithic transformation into smaller steps naturally results in multiple intermediate models, and we propose to use these models to develop and validate the transformations. Figure 1.3 shows how the transformation from model $M_{SLCO}$ to implementation $I$ is split into three steps. The refining transformations $T_R^1$ and $T_R^2$ add implementation details to model $M_{SLCO}^1$, which results in model $M_{SLCO}^3$. This model contains enough implementation details to be straightforwardly transformed to implementation $I$. Together, transformations $T_R^1$, $T_R^2$, and $T_{S2I}$ of Figure 1.3 replace the monolithic transformation $T_{S2I}$ of Figure 1.2. Analysis of model $M_{SLCO}^1$ and the intermediate models $M_{SLCO}^2$ and $M_{SLCO}^3$ provides additional insight into the transformation process that can be used for the development and validation of the transformation from SLCO models to implementations. The approach using

the monolithic transformation illustrated in Figure 1.2 does not offer this insight.

Finally, the fact that a model transformation correctly refines a single given model does not guarantee that it can be applied successfully to any possible model. Applying a model transformation to a source model and then comparing the source and target model provides useful information for the developer of the transformation, but it does not prove any properties of this transformations. To ensure that a sequence of refining model transformations leads to a correct implementation for any model it is applied to, the correctness of all model transformations in the sequence must be established.

## 1.2 Research Questions

We formulated a number of research questions aimed at resolving the problems described in Section 1.1. The central research question is as follows.

> **RQ:** *How can we improve the reliability of software that is automatically generated from high-level descriptions?*

This central research question is split into a number of more specific research questions. Each of these questions is addressed in the remainder of this thesis.

To generate software from high-level descriptions, we first need to be able to produce models that form these descriptions. During the Ideals project, we noticed that it was cumbersome to create large UML models using only graphical editors. In search of a practical solution to this problem, we formulated the following research question.

> **$\mathbf{RQ}_1$:** *How can large models for existing modeling languages be created efficiently using existing tools?*

In this thesis, we describe how software is generated from models on a high level of abstraction by first refining these models and then generating an implementation from the resulting model. The original model is refined by applying a sequence of model transformations, where the application of each transformation leads to an intermediate model. To be able to apply techniques for verification to as many of these intermediate models as possible, we formulated the following research question.

> **$\mathbf{RQ}_2$:** *How does the size and complexity of model transformations affect the verifiability of intermediate models produced by sequences of refining model transformations?*

We developed a domain-specific modeling language, called the Simple Language of Communicating Objects (SLCO), and implemented a number of transformations that refine SLCO models. To be able to prove that these transformations preserve certain desirable properties of the source model, we first need to define the formal semantics of SLCO. Before giving a formal definition of the semantics of SLCO, we wanted to experiment with a number of variations of the semantics. To do so, we implemented an executable prototype of the semantics of SLCO. The following research question is related to this prototype.

> **$\mathbf{RQ}_3$:** *What are the advantages and disadvantages of implementing an executable prototype of the semantics of a domain-specific modeling language using ASF+SDF?*

Generating reliable software by refining models is only possible if the model transformations used for the refinement preserve certain desirable properties of these models. After formally defining the semantics of SLCO, based on the aforementioned prototype, we investigated proving the correctness of a number of model transformations. This led to the following research question.

> **$\mathbf{RQ}_4$:** *Can we show that the model transformations that we implemented to refine SLCO models preserve certain desirable properties of such models?*

Although we aimed at designing a DSML that is platform independent from the start, our DSML evolved over time. To learn from our experiences in developing SLCO and to be able to apply this knowledge while designing other DSMLs, we posed the following research question.

> **$\mathbf{RQ}_5$:** *What are the main influences on the design of a DSML and the corresponding model transformations?*

Research question $\mathrm{RQ}_4$ deals with a fixed set of model transformations. We investigated how to prove the correctness of these transformations by means of proofs performed manually. In contrast, the following research question is aimed at automated verification of model transformations.

> **$\mathbf{RQ}_6$:** *Can we verify the correctness of model transformations automatically?*

## 1.3 Outline and Origin of Chapters

The remainder of this thesis is structured as follows. For each chapter that is based on an earlier publication, the origin of the chapter is given.

**Chapter 2: Integrating Textual and Graphical Modeling Languages** In this chapter, we address research question $\mathrm{RQ}_1$. We illustrate how a textual alternative for a particular type of graphical diagrams can be used to make it easier to construct large UML models, and we compare two implementations that integrate this language into the UML. This chapter is based on the following publication.

> [39] L.J.P. Engelen and M.G.J. van den Brand. Integrating Textual and Graphical Modelling Languages. *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications*, 2010. doi:10.1016/j.entcs.2010.08.035.

**Chapter 3: Simple Language of Communicating Objects** In this chapter, we introduce SLCO, the DSML that is used in the case study described in Chapter 4. The development of this language is discussed in Chapters 5, 6, and 7. This chapter is based on the following submission.

> [4] M.F. van Amstel, S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. In Vitro Development of a Domain-Specific Modeling Language. *Submitted to Science of Computer Programming*, 2012.

**Chapter 4: Exploring the Boundaries of Model Verification** In this chapter, we address research question $RQ_2$ by discussing experiments we performed to compare coarse-grained and fine-grained sequences of model transformations. This chapter is based on the following publication.

> [6] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. Using a DSL and Fine-Grained Model Transformations to Explore the Boundaries of Model Verification. *Proceedings of the Third Workshop on Model-Based Verification and Validation*, 2011. `doi:10.1109/SSIRI-C.2011.26`.

**Chapter 5: Prototyping the Semantics of a Domain-Specific Modeling Language** In this chapter, we address research question $RQ_3$. We describe the executable prototype that we implemented, and show how such a prototype can aid in the development of domain-specific modeling languages and model transformations. This chapter is based on the following publication.

> [9] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. *Proceedings of the Second International Workshop on Algebraic Methods in Model-based Software Engineering*, 2011. `doi:10.4204/EPTCS.56.5`.

**Chapter 6: Reusability and Correctness of Endogenous Model Transformations** In this chapter, we address research question $RQ_4$ by showing how the formal semantics of SLCO and formal definitions of model transformations can be used to prove the correctness of these transformations. This chapter deals with endogenous model transformations, which are transformations for which the input and output language is the same [79]. This chapter is based on the following publication.

> [10] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Reusable and Correct Endogenous Model Transformations. *Proceedings of the 5th International Conference on Model Transformation*, 2012. `doi:10.1007/978-3-642-30476-7_5`.

**Chapter 7: Evolution of a Domain-Specific Modeling Language** In this chapter, we address research question $RQ_5$. We show how SLCO has evolved over time and discuss the main influences on the design of the language. This chapter is based on the following publication.

> [5] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. An Exercise in Iterative Domain-Specific Language Design. *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010. `doi:10.1145/1862372.1862386`.

**Chapter 8: Checking Property Preservation of Refining Transformations** In this chapter, we address research question $RQ_6$. We describe a technique that makes it possible to check whether model transformations preserve certain properties. In contrast to Chapter 6, the technique described in Chapter 8 is fully automated. This chapter is based on the following publication.

[40] L.J.P. Engelen and A.J. Wijs. Checking Property Preservation of Refining Transformations for Model-Driven Development. *Technical Report, Department of Mathematics and Computer Science, Eindhoven University of Technology*, 2012.

**Chapter 9: Conclusions** This final chapter concludes this thesis. It revisits the research questions and gives directions for future research.

The research for the publications that form the basis of Chapters 2, 5, and 6 was conducted by Luc Engelen and his supervisors, and the research for the publications that form the basis of Chapters 3, 4, 7, and 8 was conducted by Luc and researchers of the Software Engineering and Technology group of the Eindhoven University of Technology. Luc served as the first author of the publications that form the basis of Chapters 2, 3, 5, and 6 and as the second author of the publication that forms the basis of Chapter 8. The publications that form the basis of Chapters 4 and 7 were written in close cooperation between Luc and Marcel van Amstel. Luc started the development of SLCO and is responsible for its initial design. He collaborated with his supervisors and Marcel van Amstel to improve and extend the language. Luc has developed the metamodels, grammars, transformations, and the proprietary tools mentioned in this thesis, except for the metamodels of NQC and Promela, and the transformation from SLCO to Promela, which were originally implemented by Marcel van Amstel and updated by Luc at a later stage. Furthermore, Luc developed the formal semantics of SLCO. All of the original work has been revised for this thesis to reflect our growing insight.

## 1.4 Suggested Method of Reading

Chapter 2 and Chapters 4 to 8 are largely self-contained and can be read independently from each other. However, to reduce duplication of information, Chapter 3 serves as an introduction to Chapters 4 to 7, and Appendix A provides short descriptions of the software tools used throughout this thesis.

# Chapter 2

# Integrating Textual and Graphical Modeling Languages

*Graphical diagrams are the main modeling constructs offered by the popular modeling language UML. Because textual representations of models also have their benefits, such as conciseness, we investigated the integration of textual and graphical modeling languages, by comparing two approaches. One approach uses grammarware, and the other uses modelware. As a case study, we implemented two versions of a textual alternative for activity diagrams, which is an example of a* surface language*. This chapter describes our surface language, the two approaches, and the two implementations that follow these approaches.*

## 2.1 Introduction

Many popular Eclipse-based modeling formalisms focus on notations that are either mainly textual or mainly graphical. Although tools exist that transform models written in a textual language to representations of those models that can be manipulated and depicted using graphical notations, the construction and manipulation of models written using a combination of both languages is not well facilitated.

The popular modeling language UML offers graphical diagrams for the construction of models. Research has shown, however, that graphical languages are not inherently superior to textual languages [91] and that both types of languages have their benefits. Therefore, we investigate the integration of textual and graphical languages to be able to exploit the benefits of both types of languages. In particular, this integration facilitates the creation of large UML models and addresses research question $RQ_1$.

> **$RQ_1$:** *How can large models for existing modeling languages be created efficiently using existing tools?*

One of the problems that arise when using two or more languages to construct one model is that parts of the model written in one language can refer to elements contained in parts written in another language. Transforming a model written in multiple languages

to a model written in one language involves introducing correct references between various parts of the model.

Existing tools are aimed at converting textual models conforming to grammars into models conforming to metamodels and vice versa [35, 61]. These tools can not transform models that consist of parts that conform to grammars as well as parts that conform to metamodels.

We use a textual alternative for activity diagrams, a textual surface language, as a case study and have implemented two versions of this language. One alternative uses tools and techniques related to grammars, and the other uses tools and techniques related to models and metamodels. The approach related to grammars transforms UML models containing fragments of behavior modeled using our surface language to plain UML models by rewriting the XMI representation of the model provided as input. We used the ASF+SDF Meta-Environment [20] to implement this approach. The approach related to models and metamodels extracts the fragments of surface language, converts them to metamodel based equivalents, transforms these equivalents to Activities, and uses these to replace the fragments in the original model. We used the openArchitectureWare platform [50, 111] to implement this approach.

The remainder of this chapter is organized as follows: Section 2.2 introduces a number of relevant concepts. A specification of the surface language we implemented, a description of its embedding in the UML, and the transformation from surface language to Activities is given in Section 2.3. The approach based on grammars is described in Section 2.4, and the approach based on models and metamodels is described in Section 2.5. A number of other applications involving the integration of textual and graphical languages, and the transformation of models constructed using multiple languages are discussed in Section 2.6. Section 2.7 provides a short description of a case study concerning the application of our surface language. Section 2.8 discusses how our work relates to earlier work. We draw conclusions and discuss future work in Section 2.9.

## 2.2 UML Activities and Surface Languages

The surface language we present is a textual alternative for the activity diagrams of the UML. In this section, we give a brief description of Activities and explain what a surface language is. We use the naming convention used by the OMG in the definition of the UML [86] when discussing concepts of the UML. This means that we use medial capitals for the names of these concepts.

### 2.2.1 UML Activities

Activities are one of the concepts offered by the UML to specify behavior. Some aspects of an Activity can be visualized in an activity diagram. The leftmost part of Figure 2.1 shows an example of such a diagram.

An Activity is a directed graph, whose nodes and edges are called ActivityNodes and ActivityEdges. There are a number of different ActivityNodes, such as ControlNodes (depicted by diamonds) and Actions (depicted by rounded rectangles), and two types of ActivityEdges, namely ControlFlows and ObjectFlows.

The informal description of the semantics of Activities states that the order in which Actions are executed is based on the flow of tokens. There are two kinds of tokens: control tokens and object tokens. ControlFlows, which are depicted by arrows

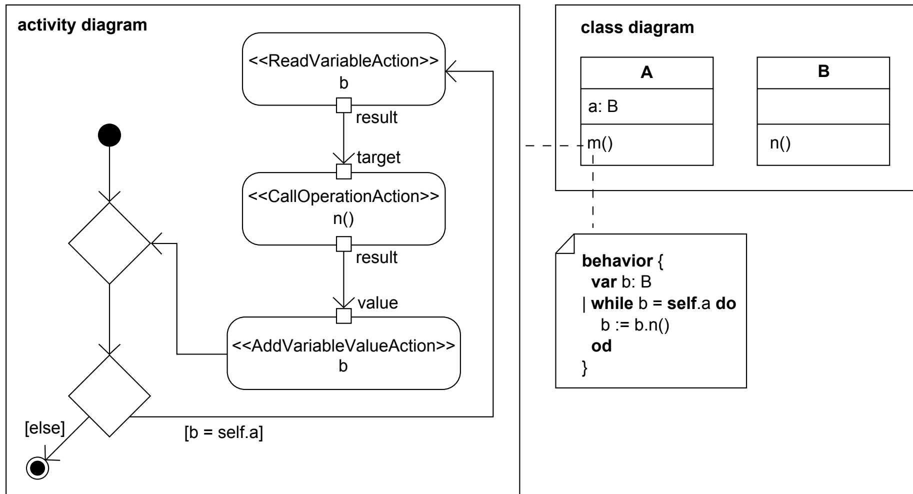



Figure 2.1: Two representations of the same behavior

connecting ActivityNodes, show how control tokens flow from one ActivityNode to the other. ObjectFlows, which are depicted by arrows connecting OutputPins and InputPins, show how object tokens flow from one Action producing an object to another Action that uses this object.

The ObjectFlows in Figure 2.1 are depicted by the arrows connecting the small rectangles on the borders of the Actions. These small rectangles are the InputPins and OutputPins of those Actions.

### 2.2.2 Surface Languages

Every model conforms to a metamodel, which defines the elements that play a role in the model. If a model conforms to a certain metamodel, each element of the model is an instance of an element in that metamodel. The UML defines a number of diagrams, which can be used to depict certain parts of a model. There are diagrams that depict the structure of a model, diagrams that depict the behavior of parts of the model, etc. These diagrams offer a graphical representation for instances of elements in the metamodel.

In the context of the UML, the term *surface language* is used to refer to a concrete syntax that offers an alternative notation for these diagrams. In our case, instead of a graphical representation, a textual representation is given for instances of elements of the metamodel. Other names for surface languages related to Activities and Actions are *surface action languages* and *action languages*.

## 2.3 Specification of the Surface Language

To define our surface language, we must specify its syntax, semantics, and embedding in the UML. The syntax of the surface language and its embedding in the UML are described below. The semantics of the language is defined implicitly by describing the transformation from behavior specified in our surface language to Activities.

### 2.3.1 Syntax

The syntax of behavior modeled in our surface language is defined as follows.

$$
\begin{array}{lll}
\mathit{SLB} & ::= & \text{“}\textbf{behavior}\text{” “\{” } [\mathit{MVD}]\ \mathit{MS} \text{ “\}”} \\
\mathit{MVD} & ::= & \text{“}\textbf{var}\text{” } \mathit{VD}\ \{\text{“;” } \mathit{VD}\}\ \text{“|”} \\
\mathit{VD} & ::= & \mathit{VN}\ \text{“:” } \mathit{TN} \\
\mathit{MS} & ::= & S\ \{\text{“;” } S\},
\end{array}
$$

where the structure of variable names *VN*, and type names *TN* is left unspecified. A description of behavior *SLB* consists of a sequence of variable declarations *MVD* and a sequence of statements *MS*. A variable declaration *VD* consists of a variable name and a type name.

The syntax of statements is defined as follows.

$$
\begin{array}{lll}
S & ::= & \text{“}\textbf{if}\text{” } E \text{ “}\textbf{then}\text{” } \mathit{MS} \text{ “}\textbf{fi}\text{”} \\
 & | & \text{“}\textbf{if}\text{” } E \text{ “}\textbf{then}\text{” } \mathit{MS} \text{ “}\textbf{else}\text{” } \mathit{MS} \text{ “}\textbf{fi}\text{”} \\
 & | & \text{“}\textbf{while}\text{” } E \text{ “}\textbf{do}\text{” } \mathit{MS} \text{ “}\textbf{od}\text{”} \\
 & | & \text{“}\textbf{return}\text{” } E \\
 & | & \mathit{SN} \text{ “(” } [\mathit{ME}] \text{ “)” “}\textbf{to}\text{” } E \\
 & | & E \text{ “.” } \mathit{ON} \text{ “(” } [\mathit{ME}] \text{ “)”} \\
 & | & E \text{ “.” } \mathit{SFN}\ [\text{“[” } N \text{ “]”}]\ \text{“:=” } E \\
 & | & \mathit{VN}\ [\text{“[” } N \text{ “]”}]\ \text{“:=” } E,
\end{array}
$$

where the structure of signal names *SN*, operation names *ON*, structural feature names *SFN*, and natural numbers *N* is left unspecified. A statement *S* can contain expressions *E* and sequences of expressions *ME*.

The syntax of expressions is defined as follows.

$$
\begin{array}{lll}
\mathit{ME} & ::= & E\ \{\text{“,” } E\} \\
E & ::= & \text{“}\textbf{create}\text{” “(” } \mathit{CN} \text{ “)”} \\
 & | & \text{“}\textbf{self}\text{”} \\
 & | & \mathit{VN} \\
 & | & E \text{ “.” } \mathit{SFN} \\
 & | & E \text{ “.” } \mathit{ON} \text{ “(” } [\mathit{ME}] \text{ “)”},
\end{array}
$$

where the structure of class names *CN* and operation names *ON* is left unspecified.

The note below the class diagram in Figure 2.1 shows an example of behavior modeled using our surface language. The behavior is equivalent to the behavior represented by the activity diagram on the left of the figure.

### 2.3.2 Embedding in the UML

We use a concept of the UML called OpaqueBehavior to embed our surface language in the UML. Listing 2.1 shows a fragment of an XMI representation of a UML model that contains an instance of OpaqueBehavior.

OpaqueBehavior uses a list of text fragments and a list of language names to specify behavior. The first list specifies the behavior in one or more textual languages, and the second list specifies which languages are used in the first list. OpaqueBehavior can be used to specify behavior using, for instance, fragments of Java code or natural language. In our case, as shown in Listing 2.1, both lists contain a single item. The first list contains

```
<packagedElement xmi:type="uml:Class" name="C">
  <ownedBehavior xmi:type="uml:OpaqueBehavior" name="b">
    <body>return self</body>
    <language>SL</language>
  </ownedBehavior>
</packagedElement>
```

Listing 2.1: Embedding in the UML of behavior modeled using a language called 'SL'

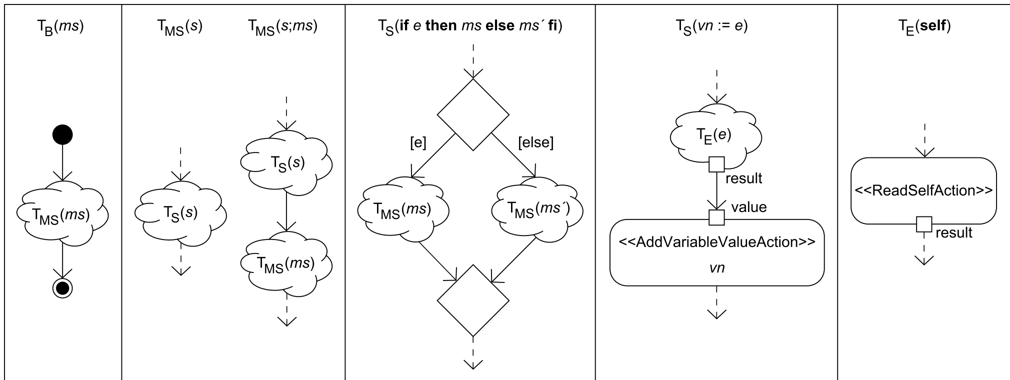



Figure 2.2: Transformations performed by the functions $T_B$, $T_{MS}$, $T_S$, and $T_E$

a specification of behavior using our surface language, and the second list indicates that we use this surface language.

We transform a UML model containing behavior modeled using our surface language to a UML model without such behavior by replacing all these occurrences of surface language embedded in OpaqueBehavior by equivalent Activities.

### 2.3.3 Transformation

As described in Section 2.3.1, behavior specified using our surface language consists of two parts: a sequence of variable declarations and a sequence of statements. The process of transforming behavior modeled using a surface language to an Activity can be divided into two steps:

1. The variable declarations are translated to UML Variables.

2. The sequence of statements is translated to an equivalent group of ActivityNodes and ActivityEdges.

Translating variable declarations to UML Variables is a trivial step, which we will not discuss. The transformation of sequences of statements to equivalent fragments of UML Activities is described informally by means of the transformation function $T_B$. The function $T_B$ uses the auxiliary transformation functions $T_{MS}$, $T_S$, and $T_E$.

Figure 2.2 gives a schematic representation of the transformations performed by these functions. The clouds and the dashed arrows in the figure indicate how the fragments are joined together to create an Activity. Each cloud in a fragment is replaced by another fragment of an Activity. An incoming dashed ActivityEdge shows how a fragment is

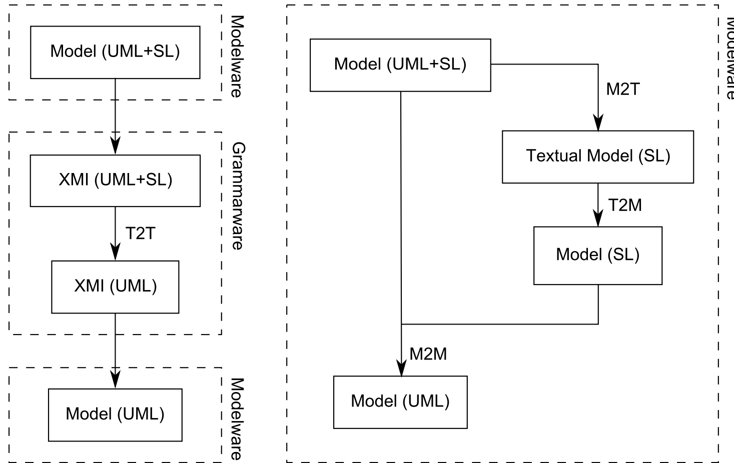



Figure 2.3: Two ways of incorporating textual languages in the UML

connected to an outgoing ActivityEdge of the containing fragment; an outgoing dashed ActivityEdge shows how a fragment is connected to an incoming ActivityEdge of the containing fragment.

The function $T_B$ creates a group of ActivityNodes and ActivityEdges that is equivalent to the sequence of statements provided as input and connects this group with an InitialNode and an ActivityFinalNode using two ControlFlows. The function $T_{MS}$ creates an equivalent group of ActivityNodes and ActivityEdges for each of the statements in the sequence provided as input and connects these groups using ControlFlows. The function $T_S$ creates a group of ActivityNodes and ActivityEdges that is equivalent to the statement provided as input. Statements and expressions that are part of the statement provided as input are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ControlFlows, for statements, or ObjectFlows, for expressions. The function $T_E$ creates a group of ActivityNodes and ActivityEdges that is equivalent to the expression provided as input. Other expressions that are part of this expression are also translated to equivalent groups of ActivityNodes and ActivityEdges. These groups are connected to the first group using ObjectFlows.

## 2.4 Grammarware

In this section, we describe the implementation of our surface language using a tool for text-to-text transformations. Tools for text-to-text transformations are often referred to as *grammarware*. We start by describing our approach in Section 2.4.1. Section 2.4.2 describes some important aspects of the implementation.

### 2.4.1 Approach

The leftmost part of Figure 2.3 gives a schematic overview of the transformation process when performed using a text-to-text (T2T) transformation. The goal of this process is to transform a UML model containing behavior specified using a surface language to a plain UML model. In the approach using grammarware, we transform models containing fragments of surface language to plain UML models by transforming the XMI [85]

representations of those models. This transformation from one textual representation to the other consists of two steps:

1. A mapping from names occurring in the model to XMI identifiers is made by traversing the parse tree of the XMI representation of the original model and storing each name and the corresponding identifier in a table.

2. The transformation described in Section 2.3.3 is performed by translating fragments of surface language to XMI representations of equivalent Activities.

The first step of the transformation makes it possible to retrieve the identifier of an element in the second step. Each element in the XMI representation of a UML model has a unique identifier. Actions that refer to other elements, such as AddVariableValueActions and CreateObjectActions, refer to these other elements using their identifiers. An AddVariableValueAction refers to a Variable using the identifier of that Variable; a CreateObjectAction refers to a Classifier using the identifier of that Classifier.

Grammarware has been a subject of research for quite some time. An advantage of using grammarware to perform this transformation is the ease of use provided by the maturity of the tools and their documentation.

A disadvantage of transforming models using a text-to-text transformation is that the models have to be exported to a textual format. Having to deal with the textual representation of a model lowers the level of abstraction of the transformation. In our case, for instance, the transformation deals with concepts of the XMI language, at a low level of abstraction, instead of concepts of the UML, at a higher level of abstraction.

### 2.4.2 Implementation

We implemented the transformation described in Section 2.3.3 following the approach described in Section 2.4.1 in the language ASF+SDF [30] using an integrated development environment (IDE) for that language, called the ASF+SDF Meta-Environment [20]. We discuss some of the details of our implementation below. The language ASF+SDF and its IDE are described in Appendix A.

An advantage of using SDF to define the syntax of our surface language is that it enabled us to combine this definition with an existing syntax definition of the XMI format, without any alterations to the definitions. This is due to the fact that context-free languages are closed under union. Because transformations implemented in ASF are syntax safe, the transformation from UML models containing fragments of surface language to plain UML models only produces results that adhere to the definition of XMI.

A disadvantage of the current implementation is that it can only parse one variant of XMI. Most tools that import or export files in the XMI format use their own interpretation of the format. These vendor specific interpretations are often incompatible with other interpretations. Because of this, our implementation is limited to XMI files produced by the UML2 plug-in of Eclipse since it can only read and produce models that adhere to the interpretation of XMI of that plug-in. A solution for this problem would be to introduce an intermediate language that serves as the starting point of a number of transformations to variants of XMI. We could then transform a model containing fragments of surface language to this intermediate language and subsequently from this intermediate language to a number of variants of XMI. The limited portability is another disadvantage of using the Meta-Environment for the implementation of our approach since it is currently only available for the Unix family of operating systems.

Listing 2.2 shows a part of the implementation in ASF of the transformation from behavior modeled using our surface language to Activities. All variable names in this listing start with a dollar sign. The listing shows that a table mapping names to identifiers, denoted by the variable $*Context*, is used both to retrieve the identifier that corresponds to a given name as well as create fresh identifiers. It also shows that every ReadVariableAction encountered in a fragment of surface language is replaced by the XMI in lines 7 to 9.

```
1  $VariableName := $ReadVariableAction,
2  variableExists($Context, $VariableName) == true,
3  $Var := getVariableId($Context, $VariableName),
4  <$Id1, $Context1> := newId($Context),
5  <$Id2, $Context2> := newId($Context1),
6  $ObjectContent* :=
7  <node xmi:type="uml:ReadVariableAction" xmi:id=$Id1 variable=$Var>
8    <result xmi:id=$Id2 />
9  </node>
10 ====>
11 statementWithResult2Action($ReadVariableAction, $Context)=
12 <$ObjectContent*, $Id2, $Id1, $Context2>
```

Listing 2.2: ASF rule that creates a ReadVariableAction

Listing 2.3 shows the part of the SDF definition that defines the syntax of the surface language statement representing a ReadVariableAction and declares the corresponding variables. Line 4 of this definition defines that a ReadVariableAction is denoted by the name of a variable, as is specified in Section 2.3.1.

```
1 sorts
2   ReadVariableAction
3 context-free syntax
4   VariableName -> ReadVariableAction
5 variables
6   "$ReadVariableAction"[0-9]* -> ReadVariableAction
```

Listing 2.3: SDF definition that defines the statement representing a ReadVariableAction

## 2.5 Modelware

This section describes the implementation of our surface language using tools for model-to-text, text-to-model, and model-to-model transformations. In this context, the term “model” refers to an instance of an explicit metamodel. Tools that can perform transformations related to models are often referred to as *modelware*. Section 2.5.1 describes our approach. Section 2.5.2 describes the most important aspects of the implementation. The tools used for the implementation are described in Appendix A.

### 2.5.1 Approach

The rightmost part of Figure 2.3 gives a schematic overview of the approach using model-to-text (M2T), text-to-model (T2M), and model-to-model (M2M) transformations within

a UML modeling tool. The process of using modelware to transform a UML model containing fragments of surface language to a plain UML model can be divided into the following steps:

1. The fragments of surface language are extracted from the original model.
2. The extracted fragments are parsed and converted to a format usable by tools for model-to-model transformations.
3. The extracted and converted fragments of surface language are translated to equivalent Activities, as described in Section 2.3.3.
4. The fragments of surface language in the original model are replaced by the Activities created in the previous step.

An advantage of this approach is that all transformations can be performed from within one and the same modeling environment. In contrast to the approach described in Section 2.4.1, no models have to be imported or exported during the transformation process.

### 2.5.2 Implementation

We used three tools for model transformation from the openArchitectureWare platform to implement the transformation described in Section 2.3.3 following the approach described in Section 2.5.1. These tools are described in Appendix A, and the implementation is described below.

We use Xpand to extract fragments of surface language from models by traversing these models. For each instance of OpaqueBehavior in a model, the string describing its behavior is stored in a text file, including the name of the OpaqueBehavior and the name of the Class it is contained in. Listing 2.4 shows the fragment of surface language extracted from the OpaqueBehavior of Listing 2.1.

```
behavior b C {
  return self
}
```

Listing 2.4: Extracted fragment of surface language

We use Xtext to parse and convert the extracted fragments of surface language to a format that is readable by Xtend. Because Xtext uses ANTLR, the class of textual representations that can be parsed is restricted to those that can be described by an LL(k) grammar. A disadvantage of using Xtext is that we had to modify our grammar for this reason.

One of the advantages of using the tools offered by the openArchitectureWare platform is their portability. The platform is a collection of plug-ins for Eclipse, and both Eclipse and these plug-ins are available on a number of different operating systems.

Listing 2.5 shows a part of the transformation implemented in Xtend from behavior modeled using our surface language to Activities. The listing shows that a new ReadVariableAction, an OutputPin, and an ObjectFlow are created by defining local variables using *let expressions*. These expressions are followed by a *chain expression*, which denotes the sequential evaluation of the expressions connected by the "->" symbols. The last two of these expressions use the ObjectFlow to connect the OutputPin of the ReadVariableAction to the InputPin of another Action.

```
Void addReadVariableAction(
   uml::Activity a, uml::Package p, surfacelanguage::Variable v,
   uml::InputPin ip
) :
   let act = new uml::ReadVariableAction :
   let op = new uml::OutputPin :
   let of = new uml::ObjectFlow :
   a.node.add(act)
-> a.edge.add(of)
-> act.setResult(op)
-> act.setVariable(v.createVariable(p))
-> of.setSource(op)
-> of.setTarget(ip)
;
```

Listing 2.5: Xtend extension that adds a ReadVariableAction to an Activity

## 2.6 Other Applications of our Approach

Our approach is not only suitable for the embedding of our textual surface language in Activities. The concept of OpaqueBehavior described in Section 2.3.2 can, for instance, also be used to embed textual languages describing behavior in other parts of the UML. Similar concepts, like OpaqueExpression and OpaqueAction, can be used to embed textual languages for other purposes than describing behavior. It is possible, for instance, to use a subset of Java as an expression language for UML *StateMachines*.

Thus far, we described how UML models combined with our surface language can be transformed to equivalent UML models. The result of the transformation described in Section 2.3.3, however, is only defined if the names used in the fragments of surface language of an input model correspond with elements that exist in the rest of the model. To check whether models meet this condition, we have implemented another version of our transformation, which performs a simple form of checking. This transformation takes a UML model containing fragments of surface language as input and transforms this into a list of error messages. The transformation traverses the model and the fragments of surface language and checks whether the names used in the statements of the surface language correspond to elements that exist in other parts of the model. If the behavior shown in the note in Figure 2.1 would refer to an attribute $self.b$, for instance, the transformation would produce a message stating that class $A$ does not have an attribute named $b$.

## 2.7 Case Study

We applied the surface language described in this chapter during a case study with an industrial partner. As part of the Ideals project [19], we investigated a straightforward transformation from UML models to formal models suited for performance analysis. To ensure the straightforwardness of this transformation, detailed UML models were required. Initially, the behavior of the components of these models was described graphically using a combination of activity diagrams and state machine diagrams, where each state machine referred to a number of activities. However, to reduce the amount of work required to produce these detailed models, we replaced some of the activity diagrams by equivalent descriptions of behavior expressed using our surface language.

| Component | Number of Activities | Number of graphical elements | Number of statements |
|---|---|---|---|
| A | 4 | 9 + 6 + 5 + 8 | 3 + 1 + 1 + 2 |
| B | 2 | 6 + 9 | 1 + 2 |
| C | 2 | 5 + 5 | 1 + 1 |
| D | 4 | 26 + 14 + 14 + 11 | 6 + 3 + 3 + 2 |
| E | 5 | 17 + 11 + 15 + 16 + 17 | 4 + 2 + 3 + 4 + 4 |
| F | 5 | 7 + 17 + 9 + 7 + 19 | 1 + 4 + 2 + 1 + 5 |
| G | 3 | 6 + 6 + 6 | 1 + 1 + 1 |

Table 2.1: Comparison of the size of graphical and textual specifications of behavior

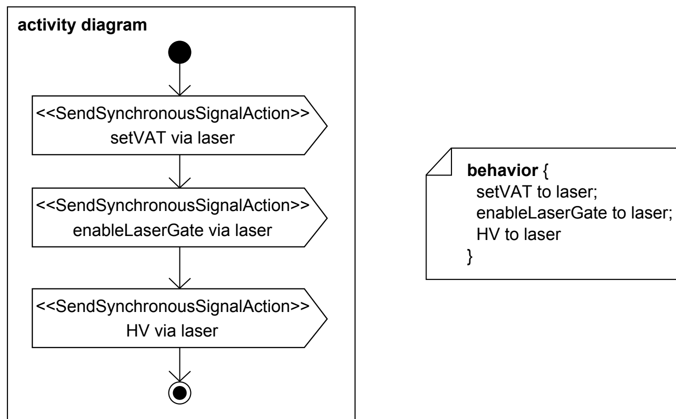



Figure 2.4: An Activity and its textual equivalent from the Ideals project

For seven of the components of such a detailed UML model, Table 2.1 shows the number of Activities required to model part of their behavior. Furthermore, the size of each of the corresponding activity diagrams is illustrated by means of the number of graphical elements contained by these diagrams. In the rightmost column of the table, the number of statements of the corresponding textual specification is given. On the left of Figure 2.4, one of the Activities of component $A$ is shown, which consists of nine graphical elements. On the right of the figure, an equivalent textual specification of this behavior is shown, expressed using our surface language.

Although most of the activity diagrams mentioned in Table 2.1 are small, such as the one in Figure 2.4, creating them showed to be cumbersome and the amount of additional information they offer in comparison to the textual specifications is limited. As mentioned above, each StateMachine is related to a number of Activities. Because of their conciseness, the textual equivalents of the Activities could be incorporated directly into the diagrams of the related state machines, thus reducing the number of diagrams.

## 2.8 Related Work

We chose to design and implement a new surface language instead of implementing a design proposed by others. Section 2.8.1 describes two existing proposals for surface languages and indicates why we decided not to implement either of them. There are many

alternatives for the languages we used to implement our surface language. Section 2.8.2 lists a number of alternatives for the grammarware we used and Section 2.8.3 lists a number of alternatives for the modelware we used. Section 2.8.4 describes another approach for integrating textual and graphical modeling languages.

### 2.8.1 Surface Languages

Dinh-Trong, Ghosh, and France propose an Action Language based on the syntax of Java [34]. We decided not to implement their Action Language because their definition of the language contains a number of primitive types and Java constructs whose relation to the UML is not specified. Other important features of their language are that parameters that serve as input or output of an Activity and attributes with multiplicity greater than one are not taken into account.

Haustein and Pleumann propose a surface language that is an extension of the OCL [51, 87]. They embed OCL expressions in their language by adding an Action to the UML that evaluates an OCL expression and returns the resulting value. We took a different approach because we wanted to design and implement a simple alternative for activity diagrams that did not rely on or incorporate other languages. Incorporating an expression language like the OCL in our language would introduce a large number of language constructs that have no relation to our primary interest, which is the specification of behavior.

### 2.8.2 Grammarware

SDF is based on SGLR, a scannerless generalized LR parser [110]. As an alternative to using SDF, SGLR can be used directly to parse textual representations of models. However, this requires either manual creation of the parse tables that SGLR takes as input or generating them from language descriptions formalized using a custom syntax definition formalism. Since SGLR can parse arbitrary languages with a context-free syntax and context-free languages are closed under union, multiple syntax definitions can be combined into one without any modifications to the original syntax definitions, as is the case for SDF.

Other common tools used for parsing, such as ANTLR, JavaCC [66], and YACC [58], can also be used to parse textual representations of models. They pose more restrictions on the grammars used for the description of the textual representations, however, since the grammars need to be of the LALR or the LL class.

After parsing the textual representations of models, the resulting parse trees have to be transformed. Besides using special purpose transformation tools, generic programming languages can be used to manipulate the parse trees. The source transformation language TXL [28] is an example of a special purpose language. Paige and Radjenovic [89], and Liang and Dingel [76] have experimented with TXL in the context of model transformation. Although their research also deals with using grammarware for transformations related to models, it differs from ours because it does not focus on the integration of text-based and metamodel-based languages.

Stratego/XT provides an alternative for ASF+SDF [22]. It is a language and toolset for program transformation that also uses SDF for parsing. It offers programmable rewrite strategies, which allow its users to define the order in which rewrite rules are applied. In ASF+SDF, the user has no control over the order of application. In contrast to ASF+SDF, Stratego/XT is not supported by an IDE.

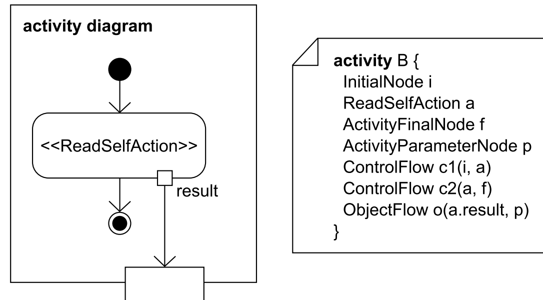



Figure 2.5: An activity diagram and a straightforward textual equivalent

### 2.8.3 Modelware

TCS [61] is an alternative for Xtext. It is suited for both text-to-model and model-to-text transformations and uses one specification to define the transformations in both directions. In the case of TCS, the main constructs are called templates. These templates are similar to the rules of Xtext; each template specifies the textual representation of an instance of an element of the metamodel.

Figure 2.5 illustrates how the resulting languages differ from our surface language in case a straightforward mapping, like those offered by Xtext and TCS, is used without additional transformations. The behavior shown in Figure 2.5 is equivalent to the behavior shown in Listing 2.4. The description of behavior shown in Figure 2.5 is much more wordy than that of Listing 2.4, even for such a trivial example.

There are many languages for model transformation, including QVT [84], ATL [60], and Epsilon [67]. Since our approach does not rely on any specific properties of Xtend, each of these transformation languages can replace Xtend in our implementation.

### 2.8.4 Embedding Textual Modeling into Graphical Modeling

Scheidgen's approach for integrating textual and graphical modeling languages [102] is based on the fact that Eclipse uses the *Model View Controller* pattern [96]. A mapping from textual notation to metamodel elements is used to generate a model from a textual representation of that model and vice versa. This custom textual notation and the graphical notations provided by Eclipse provide independent *Views* for the same *Model*. The *Controller* is used to modify the underlying model without interfering directly with the other views. The embedded text editor contained in the implementation of this approach offers syntax highlighting and code completion. Similar to Xtext and TCS, the language describing mappings from textual notation to metamodel elements offers only straightforward mappings, which makes it less flexible than the approach using grammarware.

## 2.9 Conclusions and Future Work

In this chapter, we addressed research question $RQ_1$ by investigating two approaches for the integration of textual and graphical modeling languages. To create large, detailed UML models efficiently, we implemented a textual surface language as an alternative for activity diagrams. We described this surface language, the two approaches, the

implementations that follow these approaches, and a number of related applications. The approach using grammarware transforms models containing fragments of surface language to plain models by rewriting the XMI representations of these models. The approach using modelware extracts fragments of surface language from a model, converts these fragments to a representation based on metamodels, transforms them to equivalent Activities, and replaces the original fragments with the equivalent Activities. The approaches we presented are not limited to the transformation of models to equivalent models. We also implemented a transformation that transforms models containing fragments of surface language into a list of error messages, thus providing a simple form of checking.

The research presented in this chapter did not focus on studying the suitability of textual surface languages as a method for the efficient creation of large, detailed models. However, experiments with a case study related to the Ideals project [19] showed that the surface language described in this chapter does provide a convenient way of creating such models for the UML. By replacing activity diagrams with fragments of surface language, the number of diagrams could be significantly reduced, without reducing the understandability of the model of the case study.

Both approaches and the corresponding implementations have their advantages and disadvantages when applied to integrate a textual language into an existing modeling language. The main advantages of the approach that uses grammarware are the flexibility offered by the syntax definition formalism and the ease of use provided by the maturity of the tools and their documentation. A downside of this approach is that dealing with the XMI representation of models lowers the level of abstraction of the transformations related to the approach. An advantage of the approach that uses modelware is that all of the aforementioned operations related to this approach can be performed from within one modeling environment. A disadvantage of the current implementation of this approach is that the available tools pose more restrictions on the grammar of the language we embed, in comparison to the approach using grammarware. Our approaches provide advantages over the approaches described in Section 2.8 because they both offer a more complex mapping from textual representations to metamodel elements, which can be used to obtain simpler textual representations. The fact that the implementation using grammarware poses less restrictions on the syntax of the textual language is also an advantage over these approaches.

The current implementation that uses grammarware can parse only one variant of XMI, but a future extension that introduces an intermediate language could pose a solution for this shortcoming. Investigating the use of more advanced parsing technology as a basis for the modelware tools is another promising direction for future research.

## Chapter 3

# Simple Language of Communicating Objects

*The Simple Language of Communicating Objects (SLCO) is a small domain-specific modeling language for the specification of systems consisting of objects that operate in parallel and communicate with each other. Via a number of model transformations, SLCO models can be simulated, executed, and verified. In this chapter, we describe the language itself, the languages used for simulation, execution, and verification, and the model transformations related to the language. Additionally, we describe the implementation of the language and its transformations. This chapter serves as an introduction for Chapters 4 to 7. Chapter 4 uses SLCO for a case study, and Chapters 5, 6, and 7 describe its development. In this chapter, we provide an informal description of the semantics of SLCO. A formal semantics is presented in Appendix B.*

## 3.1 Metamodel

An SLCO model consists of a number of classes, objects, and channels, as shown by the partial metamodel in Figure 3.1. Objects are instances of classes. A class describes the structure and behavior of its instances. It has ports and variables that define the structure of its instances and state machines that describe their behavior. It is possible to specify the initial values of variables. If no initial value is specified, integer variables are initialized to 0, Boolean variables are initialized to **true**, and string variables are initialized to the empty string. The variables of a class are global variables in the sense that they can be used by all state machines that are part of the class. Ports are used to connect channels to objects, and each port is connected to at most one channel. The state machines that are part of a class can only send and receive signals via the ports of this class. In case a class specifies that its instances consist of multiple state machines, these state machines operate in parallel.

A state machine consists of variables, states, and transitions. In contrast to the variables of a class, the variables of a state machine are local variables because they can only be used by the state machine that contains them. SLCO offers two special types of states: initial states and final states. A state machine starts in its initial state, and an

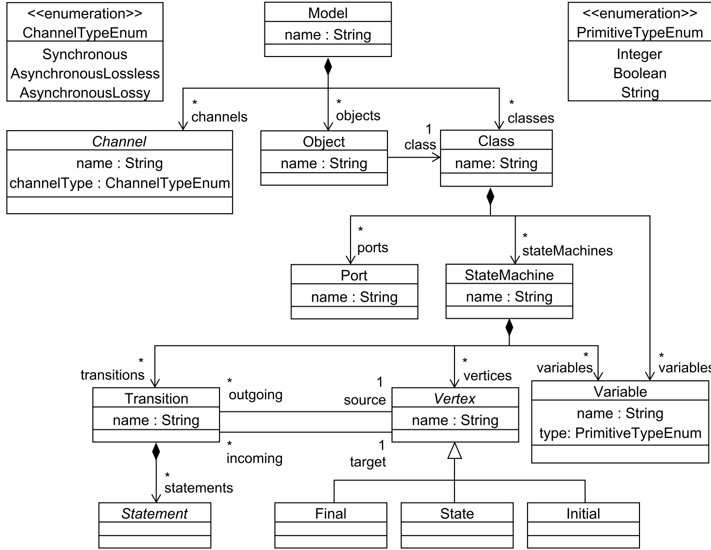



Figure 3.1: Main constructs of SLCO

SLCO model has successfully terminated when all its constituting state machines have reached a final state. Each state machine has exactly one initial state and can contain any number of ordinary and final states. A transition has a source and a target state, and is associated with a finite, ordered sequence of statements. Each statement is either blocked or enabled, and a transition is enabled if its sequence of statements is empty or the first statement of its sequence is enabled. Otherwise, it is blocked. Taking an enabled transition from its source state to its target state leads to the execution of the associated statements. If one of these statements is blocked, then its execution is halted until it becomes enabled. The execution of a single statement is atomic, but the execution of a number of statements is not. This means that the execution of statements related to a given transition can be interleaved by the execution of statements related to a transition of a concurrent state machine. If a transition has multiple outgoing transitions that are enabled, one of them is taken non-deterministically.

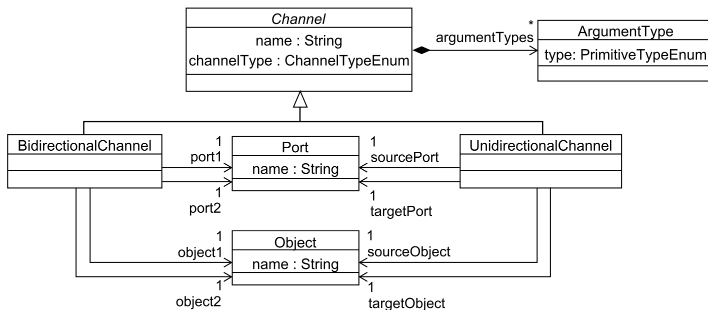



Figure 3.2: Channels in SLCO

Objects communicate with each other via channels, which are either bidirectional or unidirectional. SLCO offers three types of channels: synchronous channels, asynchronous,

lossy channels, and asynchronous, lossless channels. Each asynchronous channel is implicitly associated to one or two one-place buffers. A unidirectional channel is associated to one buffer, and a bidirectional channel is associated to two buffers, one for each direction. Signals can be sent over an asynchronous, lossless channel in a certain direction if the buffer associated to that direction is empty. If a buffer is not empty, statements that send signals over this buffer block. Signals can always be sent over asynchronous, lossy channels. Because these channels are lossy, however, some signals sent over these channel are not stored in the corresponding buffer. If the buffer corresponding to a channel already contains a signal and another signal is sent over this channel, the existing signal is replaced with the new signal. If the buffer associated to a channel is empty, the signal reception statements that receive signals via this channel are blocked. If the buffer associated to a channel contains a signal and a matching signal reception statement is executed, the signal is removed from the buffer and received by the state machine executing the signal reception statement. A channel can only be used to send and receive signals with a certain signature, which defines the number of arguments of a signal and the types of these arguments. The part of the SLCO metamodel concerning channels is shown in Figure 3.2.

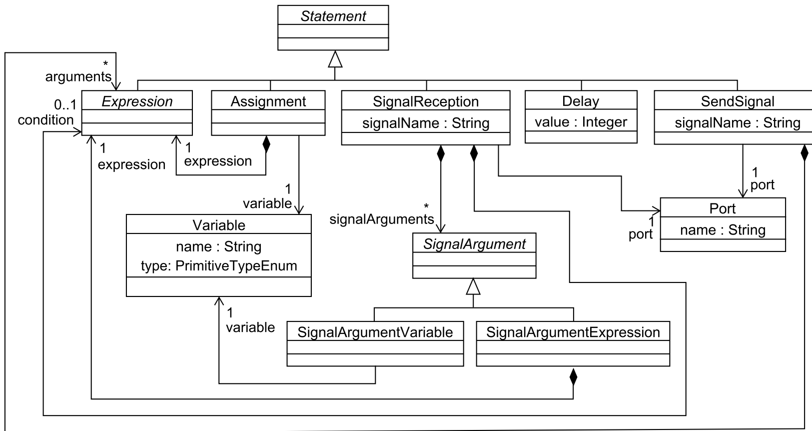



Figure 3.3: Statements in SLCO

SLCO offers five types of statements, as shown in Figure 3.3. A Boolean expression represents a statement that blocks the transition from a source to a target state until the expression evaluates to **true**. The part of the SLCO metamodel concerning expressions is shown in Figure 3.4. A delay statement blocks a transition until a specified amount of time measured in milliseconds has passed. A transition with a (conditional) signal reception statement is enabled if a signal with appropriate arguments is received via the indicated port and the optional condition holds. There are two ways of specifying that a signal reception is conditional. First, expressions given as arguments of a signal reception specify that only signals whose argument values are equal to the corresponding expressions are accepted. Second, only those signals are accepted for which the optional condition of a signal reception evaluates to **true**. This condition is a Boolean expression that may refer to the arguments of the signal that is offered via the port. Besides these statements, SLCO also offers statements for assigning values to variables and for sending signals via ports.

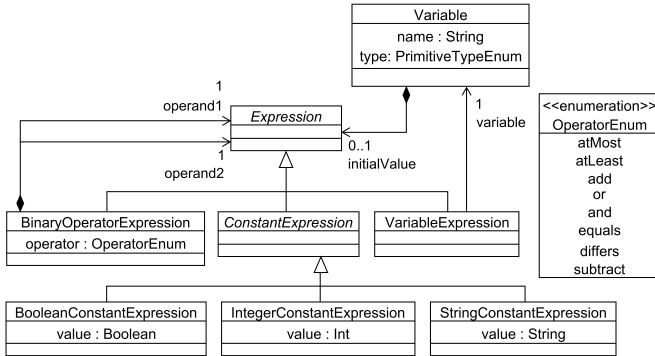



Figure 3.4: Expressions in SLCO

## 3.2 Concrete Syntax

SLCO has both a textual and a graphical concrete syntax. The graphical concrete syntax consists of communication diagrams, structure diagrams, and behavior diagrams. The diagrams in Figures 3.5, 3.6, and 3.7 show an example of an SLCO model using the graphical syntax. Below, we describe the model shown in these figures.

The communication diagram in Figure 3.5 shows two objects $p$ and $q$ that communicate over an asynchronous, lossless channel *c1*, an asynchronous, lossy channel *c2*, and a synchronous channel *c3*. Asynchronous, lossless channels, such as *c1*, are denoted by dashed lines, asynchronous, lossy channels, such as *c2*, are denoted by dotted lines, and synchronous channels, such as *c3*, are denoted by solid lines. The diagram shows that both objects have three ports. The ports of object $p$ are named *In1*, *In2*, and *InOut*, and the ports of object $q$ are named *Out1*, *Out2*, and *InOut*. Arrowheads indicate the directionality of channels: unidirectional channels have one arrowhead and bidirectional channels have two. Object $q$ can only send signals over channel *c1* via its port *Out1*, for example, and object $p$ can only receive signals sent over channel *c1* via its port *In1*. Channel *c3*, however, can be used for communication in both directions. Channel *c1* can only be used to send and receive signals with a Boolean argument, channel *c2* is restricted to signals with integer arguments, and channel *c3* is restricted to signals with string arguments. The figure also shows that object $p$ is an instance of class $P$ and that object $q$ is an instance of class $Q$.

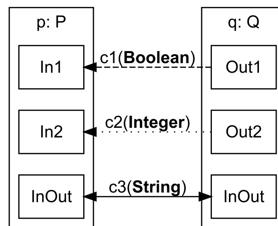



Figure 3.5: Objects, ports, and channels of an SLCO model

The structure diagram in Figure 3.6 shows the structure of classes $P$ and $Q$. Class $P$ consists of three state machines (*Rec1*, *Rec2*, and *SendRec*), and class $Q$ consists of one state machine (*Com*). Furthermore, it shows that class $P$ comprises an integer variable $m$

and that state machines *SendRec* and *Com* comprise a string variable $s$. The initial value of variable $m$ is 0. The initial values of the other variables are not specified, which means that they are initialized to empty strings as described above.

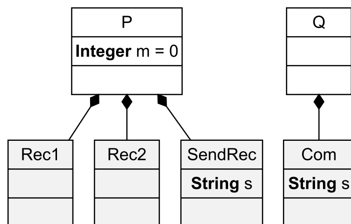



Figure 3.6: Classes, state machines, and variables of an SLCO model

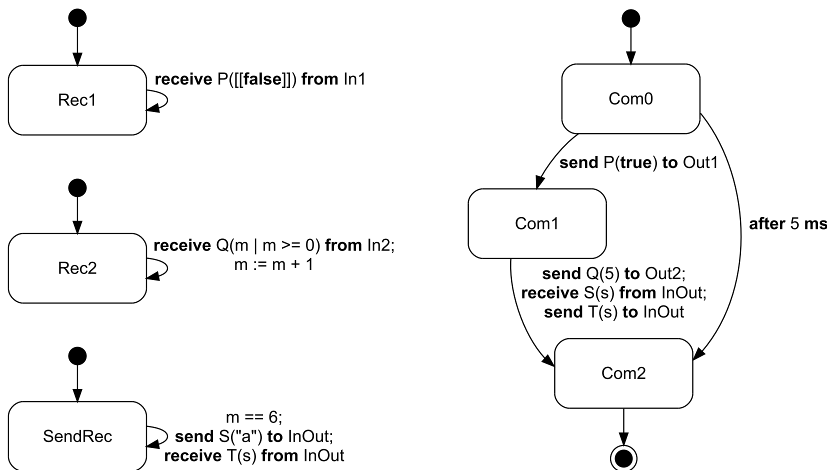



Figure 3.7: State machines of an SLCO model

The behavior diagram in Figure 3.7 shows the four state machines of our example model. The three state machines on the left of this figure specify the behavior of object $p$, and the state machine on the right specifies the behavior of object $q$. Initially, all of these state machines are in their initial states, which are named *Rec1*, *Rec2*, *SendRec*, and *Com0*. The fact that these states are initial states is indicated by an incoming arrow from a solid black dot. The black dot itself does not represent a separate state.

When state machine *Com* makes the transition from state *Com0* to state *Com1*, it executes the statement **send** $P($**true**$)$ **to** *Out1*. The signal sent by object $q$ is never received by object $p$, however, because the statement **receive** $P([[$**false**$]])$ **from** *In1* of state machine *Rec1* can only receive signals if their argument equals **false**. The double square brackets in the signal reception statement are used to distinguish expressions from variables. For instance, **receive** $S(v)$ **from** $P$ represents a statement that receives signals named $S$ from port $P$ and stores the value of the received argument in variable $v$. In contrast, the statement **receive** $S([[v]])$ **from** $P$ only receives those signals named $S$ from port $P$ whose argument is equal to the value of variable $v$.

After state machine *Com* has been in state *Com0* for 5 ms, the transition from state *Com0* to final state *Com2* is enabled. This means that after 5 ms have passed while

state machine *Com* is in state *Com0*, it can non-deterministically choose to send a signal or terminate. The fact that state *Com2* is a final state is indicated by an outgoing arrow to a circled black dot. Also in this case, the circled black dot itself does not represent a separate state.

While making the transition from state *Com1* to state *Com2*, a signal $Q(5)$ is sent to port *Out2* first. Then, after a signal $S$(s) is received from port *InOut* and a signal $T$(s) is sent to the same port afterwards, state *Com2* is reached. Both ports named *InOut* are connected to a synchronous channel, which means that communication via these ports is synchronous. A statement that sends signals via these ports can only be executed when the object on the other side of the channel is able to accept the signal that is being sent. If there is no matching signal reception for a given statement that sends a signal, the latter statement blocks.

The conditional signal reception statement **receive** $Q(m \mid m >= 0)$ **from** *In2* specifies that state machine *Rec2* will only accept a signal $Q$(m) if the condition $m >= 0$ holds. After receiving such a signal, the value of global variable $m$ is incremented.

The first statement of the transition of state machine *SendRec* blocks until the value of global variable $m$ equals 6. Once this condition holds, a signal is sent via port *InOut*, after which the state machine waits for the reception of a signal via the same port.

```
model CoreWithTime {
  classes
    Q {
      ports Out1 Out2 InOut
      state machines
        Com {
          variables String s
          initial Com0 state Com1 final Com2
          transitions
            InitialToState from Com0 to Com1 {
              send P(true) to Out1
            }
            ...
        }
    }
    ...
  objects p: P q: Q
  channels
    c1(Boolean) async lossless from q.Out1 to p.In1
    c2(Integer) async lossy from q.Out2 to p.In2
    c3(String) sync between p.InOut and q.InOut
}
```

Listing 3.1: Part of a textual SLCO model

As mentioned above, SLCO also has a textual concrete syntax. Listing 3.1 shows a part of the example model using the textual syntax.

## 3.3 Target Languages

SLCO models can be simulated, executed, and verified by transforming them to equivalent models and implementations in a number of languages. Before describing the transformations from SLCO to these languages, we discuss the target languages themselves.

### 3.3.1 POOSL

We use the Parallel Object-Oriented Specification Language (POOSL) [108], a formal modeling language for simulation and performance analysis, for simulation of SLCO models. The behavioral part of POOSL is based on the formal language CCS [81], and the part for modeling data is based on traditional object-oriented languages. A POOSL model consists of a set of concurrent processes connected by channels. These processes can communicate by exchanging synchronous messages via these channels. POOSL is supported by two tools: SHESim and Rotalumis. SHESim offers interactive simulation of POOSL models using its built-in POOSL interpreter. Rotalumis is a command-line tool that can simulate POOSL models at high speed by compiling them to byte code that can be executed on a virtual machine.

### 3.3.2 NQC

To execute SLCO models, an implementation platform is required. We chose to use the Lego Mindstorms[1] platform for this purpose. The key part of this platform is a programmable controller called RCX. This RCX has an infrared port for communication and is connected by wires to sensors and motors for interaction with its environment. We deliberately opted for the outdated RCX controller, instead of the newer and more advanced NXT controller, to investigate the strength of our transformational approach when dealing with a primitive execution platform. The language we use to program these programmable controllers is called Not Quite C (NQC) [13]. NQC is a restricted version of C, combined with an API that provides access to the various capabilities of the Lego Mindstorms platform, such as sensors, outputs, timers, and communication via the infrared ports.

### 3.3.3 Promela

Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [26]. We use the model checker Spin [55] for verifying our models. Spin can, among others, check a model for deadlocks, unreachable code, and determine whether it satisfies a Linear Temporal Logic (LTL) property [95]. LTL is used to express properties of paths in a finite-state representation of the state space of a system. The input language for Spin is Promela. Promela has constructs for modeling selections and loops, based on Dijkstra's guarded commands, and primitives for message passing between processes over channels, either using queues or handshaking. This enables modeling of both asynchronous and synchronous communication, respectively. The syntax of expressions and assignments in Promela is similar to that of C.

## 3.4 Semantic Gaps and Platform Gaps

There are a number of differences between SLCO, POOSL, NQC, and Promela in terms of the language constructs they offer. These differences are often referred to as semantic gaps [8]. Furthermore, the Lego Mindstorms execution platform has practical limitations that poses restrictions to implementations in NQC. These restrictions form platform gaps because they do not hold for models specified using the other languages. We identified a

[1] `http://mindstorms.lego.com/`

number of gaps between SLCO and the target languages and platforms, which are shown in Table 3.1. To successfully transform an SLCO model into an equivalent model or implementation in one of the other languages, these gaps have to be bridged.

| | **NQC** | **POOSL** | **Promela** | **SLCO** |
|---|---|---|---|---|
| **(A)synchronous communication** | asynchronous | synchronous | both | both |
| **Reliability of communication** | unreliable | reliable | reliable | both |
| **Support for string constants** | no | yes | symbolic names for integer constants | yes |
| **Connectivity for communication** | broadcast | point-to-point | point-to-point | point-to-point |
| **Number of objects** | limited | $\infty$ | $\infty$ | $\infty$ |

Table 3.1: Language and platform characteristics

Each column lists the characteristics of one of the four languages and the corresponding platform. The first row indicates whether communication is synchronous or asynchronous. In case communication is synchronous, both the sender and receiver of a signal need to be available before a signal can be sent. In this way, sender and receiver synchronize on communication. In case communication is asynchronous, a sender can send a signal and proceed with its execution even though the receiver is not yet ready to receive the signal. The second row indicates whether communication over channels is reliable. In case a channel is reliable, which is also referred to as lossless, a signal that is sent will always arrive at the receiving end. In case a channel is unreliable, which is also referred to as lossy, a signal that is sent may get lost. The third row indicates whether a language supports string constants. Although Promela has no support for strings, it offers an enumerated type that allows representing numeric constants using symbolic names, which can be regarded as a restricted form of string constants. The fourth row shows whether signals are broadcasted or sent using point-to-point communication. When signals are broadcasted, each signal can be received by multiple objects. In the case of point-to-point communication, however, signals are sent from one object to exactly one other object. The fifth row lists the amount of objects that can be instantiated simultaneously. In POOSL, Promela, and SLCO, this amount is unlimited. For Lego Mindstorms, however, this number is limited in practice. Because every object should be deployed on an RCX, the amount of concurrent objects is bounded by the available number of RCX controllers.

## 3.5 Model Transformations

Two types of model transformations are applied to transform SLCO models to models or implementations in one of the target languages: endogenous transformations and exogenous transformations. The input and output language of an endogenous model transformation is the same, whereas exogenous model transformations transform models from one language to another language [79]. The endogenous transformations are used to refine SLCO models by bridging the semantic and platform gaps. After all gaps have been bridged, the straightforward exogenous transformations can be applied.

### 3.5.1 Endogenous Transformations

For each of the semantic gaps and platform gaps described in Section 3.4, we implemented an endogenous model transformation that bridges this gap. To keep these transformations simple, they can only be applied to models adhering to certain conditions. In addition to the transformations that bridge the gaps, we implemented transformations that refine models to ensure that they adhere to the aforementioned conditions.

#### 3.5.1.1 Synchronized Communication over Asynchronous Channels

To bridge the semantic gap between languages offering synchronous communication and languages that do not, we implemented two transformations. Both transformations take an SLCO model and a synchronous channel as input, and produce a model in which this channel is replaced by an asynchronous, lossless channel and the objects that communicate over this channel are adapted such that they communicate asynchronously.

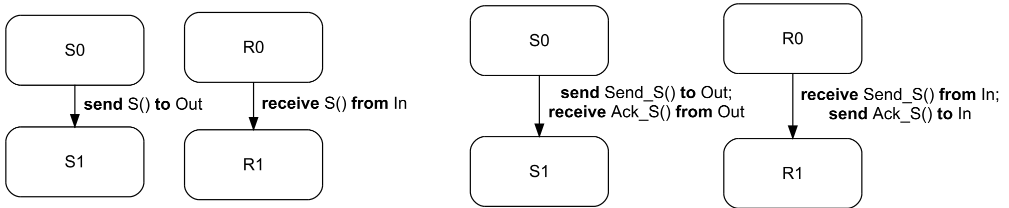



Figure 3.8: State machines before and after applying the simple version of $T_{as}$

The first transformation is simple, but can only be applied to models that do not contain states with multiple outgoing transitions if one of these transitions starts with a statement that sends a signal over the synchronous channel. The transformation ensures that the behavior of the model is still as desired by adding acknowledgment signals for synchronization. Whenever a signal is sent, the receiving party sends an acknowledgement indicating that the signal has been received. The sending party waits until it receives this acknowledgement. In this way, synchronization is achieved. On the left of Figure 3.8, two partial state machines are shown that send and receive a signal $S$. Initially, ports *In* and *Out* are connected by a synchronous channel. After transformation, acknowledgements are added, as shown on the right of the figure, and the synchronous channel is replaced with an asynchronous, lossless channel. This transformation is described in more detail in Appendix C, and its correctness is discussed in Chapter 6 and Appendix D.

The second transformation can be applied to any SLCO model, but is more complex. The partial state machine on the left of Figure 3.9 shows an example of a situation where the transformation described above cannot be applied. State *S0* has multiple outgoing transitions, and one of these transitions starts with a statement that sends a signal. In such situations, a more complex protocol has to be applied to ensure that the behavior of the model before and after transformation is the same, apart from the communication over the channel that is replaced. Also in this example, ports *In* and *Out* are initially connected by a synchronous channel, which is replaced by an asynchronous, lossless channel. In Figure 3.9, the second partial state machine from the left shows how the sending state machine is affected by the transformation, and the third partial state machine from the left shows how the receiving state machine is affected. Additionally, two other state machines are added to the model, which are shown on the right of

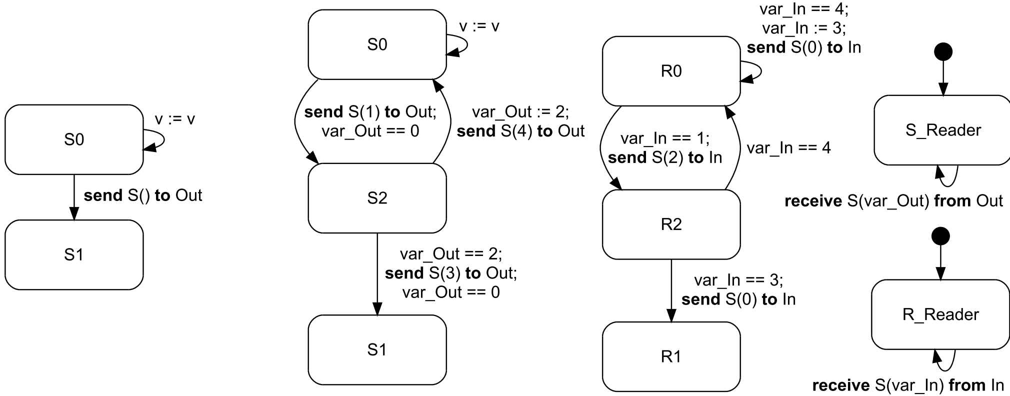



Figure 3.9: State machines before and after applying the general version of $T_{as}$

Figure 3.9. State machine *S_Reader* is added to the object that sends signals, and state machine *R_Reader* is added to the object that receives signals. Together with these state machines, the sender and receiver implement a protocol that ensures that states *S1* and *R1* are only reached if the communication between the sender and receiver was successful. Furthermore, as long as the sender and receiver are unable to communicate, all enabled outgoing transitions of states *S0* and *R0* can be made. Appendix C provides a more detailed description of this transformation.

The protocol that is employed by this transformation is not straightforward. Therefore, we used the state-space generator for SLCO described in Chapter 5 for feedback during its development. Informally, the protocol consists of the following steps. Initially, the value of variable *var_Out* is equal to 0, and the value of variable *var_In* is equal to 3. First, the sending object sends a signal $S(1)$ to indicate that it wants to communicate. It can proceed to state *S2* if all previous signals have been acknowledged by the receiving object, which is the case if variable *var_Out* is equal to 0. The signal sent by the sending object is received by state machine *R_Reader*, and the value of its argument is stored in *var_In*. Once the receiving object is informed of the intent of the sending object by means of the execution of statement *var_In == 1*, it sends a signal $S(2)$ to indicate that it is ready to communicate. Once this signal has been received and the value of its argument has been stored in variable *var_Out* by state machine *S_Reader*, the sending object may choose to complete the communication by sending signal $S(3)$. Alternatively, it may choose to cancel the communication by sending signal $S(4)$. Upon receiving one of these signals, the receiving object can take the transition to state *R1* if the communication has been completed successfully, or take the transition to state *R0* if it has been canceled. Either way, it acknowledges the reception of the signal of the sending object by sending a signal $S(0)$. The state machines *S_Reader* and *R_Reader* ensure that the statements that send signals cannot be blocked, by emptying the buffers associated to the channels continuously. It is possible that the sending object sends signal $S(4)$ after sending signal $S(1)$, while the receiving object remains in state *R0*. The self-loop on state *R0* ensures that an acknowledgement is also sent in this situation.

In the remainder, the simple version of this transformation is referred to as $T^S_{as}$, and the general version is referred to as $T^G_{as}$. In cases where it is not relevant which of these two transformations is applied, the abbreviation $T_{as}$ is used.

#### 3.5.1.2 Lossless Communication over a Lossy Channel

Transformation $T_{ll}$ implements lossless communication over a lossy channel by introducing auxiliary objects that implement a concurrent version of the Alternating Bit Protocol (ABP) [12] known as the Concurrent Alternating Bit Protocol (CABP) [11]. This transformation is only applicable to unidirectional channels that are used to communicate signals named *Signal* and whose only argument is a string.

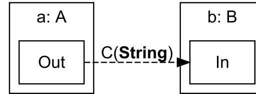



Figure 3.10: Two objects communicating over an asynchronous, lossless channel

Figure 3.10 shows a model consisting of two objects ($a$ and $b$) that communicate over an asynchronous, lossless channel ($C$). After transformation, channel $C$ is replaced by the channels *c1* to *c6* and four objects that implement the CABP, as shown in Figure 3.11. Object $a$ is connected to an object named *sender*, which communicates over an asynchronous, lossy channel *c2* with an object named *receiver*. The object named *receiver* is in turn connected to the object $b$. After transformation, objects $a$ and $b$ communicate with each other via the aforementioned objects, instead of directly. Objects $a$ and $b$ are connected to these objects by synchronous channels. After receiving a signal from object $a$, object *sender* repeatedly sends this signal over channel *c2* until it receives an acknowledgement from object *ar*, to which it is connected via the synchronous channel *c6*. After receiving a signal over channel *c2*, object *receiver* forwards this signal to object $b$ and instructs object *as* to continuously acknowledge the reception of this signal. Object *as* does this by continuously sending signals over the asynchronous, lossy channel *c5*. The acknowledgement sent by object *as* contains a two-valued argument that is used by object *ar* to assess whether a particular acknowledgement signal was already received before. Once object *ar* has received a new acknowledgement, it notifies object *sender*. After receiving such a notification, object *sender* is able to receive a new signal from object $a$ and transmit this signal over channel *c2*.

In Figure 3.12, the four state machines are shown that specify the behavior of the four objects that implement the CABP. To show which state machine is part of which class, the names of the states have been chosen such that they reflect the names of the classes and objects.

#### 3.5.1.3 Adding Delays to Transitions

Transformation $T_{time}$ takes a model and a set of transitions as input, and adds delay statements to these transitions. This transformations is used to control the frequency of the acknowledgments sent by the objects implementing the CABP. Because it reduces the number of signals that are sent, it also reduces the number of collisions between messages sent via infrared on the Lego Mindstorms platform.

#### 3.5.1.4 Replacing Strings by Integers

Transformation $T_{int}$ replaces each string constant in an SLCO model with a unique integer constant and changes the type of all string variables and arguments to integer. This transformation deals with the fact that NQC does not offer strings.

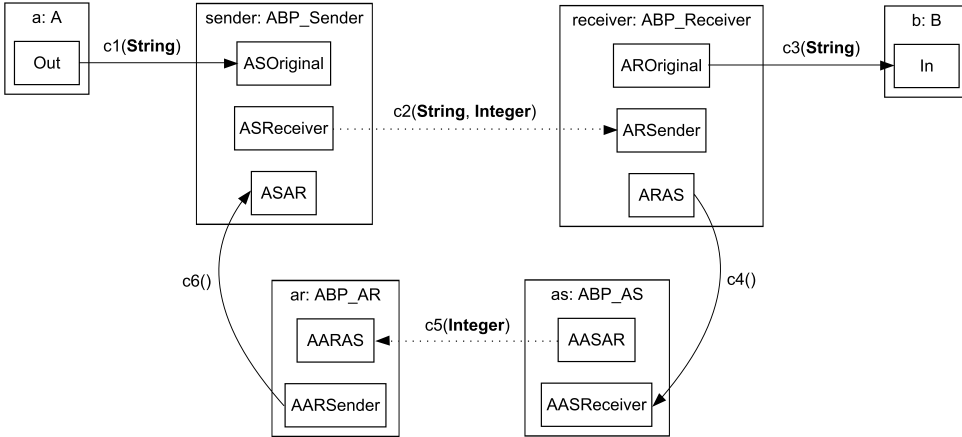



Figure 3.11: Two objects that communicate via the CABP

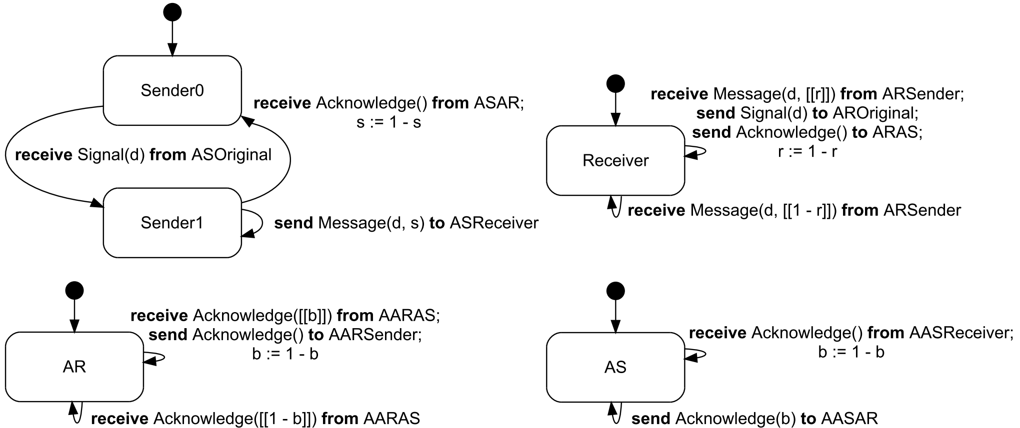



Figure 3.12: Four state machines implementing the CABP

#### 3.5.1.5 Making the Sender of a Signal Explicit

When multiple objects broadcast signals with the same name and number of arguments over the same medium, the receiving object cannot determine the origin of such a signal. This situation arises when multiple RCX controllers communicate with each other, because they communicate by broadcasting messages via infrared. To enable a receiving controller to determine the origin of each signal it receives, transformation $T_{ic}$ can be applied to a model. This transformation takes a model and a set of channels as input, and adds an index to all signal names that identifies the channel over which these signals are sent.

#### 3.5.1.6 Reducing the Number of Objects

Transformation $T_{merge}$ merges multiple objects into one object. Given a model and a set of objects, it creates a new object that contains all the variables, ports, and state machines contained by the objects provided as input. By reducing the number of objects in a model, it bridges the corresponding gap between SLCO and NQC. If any of the objects that are

being merged communicate over synchronous channels, then this form of communication is replaced by communication using shared variables. Transformation $T_{merge}$ is only applicable to objects that satisfy the following condition: each pair of state machines that are part of two communicating objects must communicate over a unique unidirectional, synchronous channel.

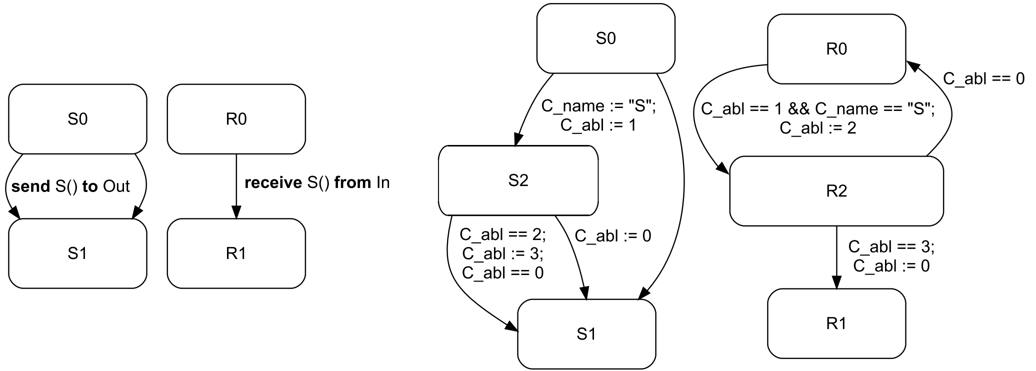



Figure 3.13: Two state machines before and after merging objects

Figure 3.13 shows how communication over a synchronous channel is replaced by communication using shared variables. The two partial state machines on the left of the figure are part of two separate objects and communicate with each other by sending and receiving signals over a synchronous channel that connects ports *In* and *Out*. After merging these two objects, the state machines are adapted as shown on the right of the figure and communicate using the shared variables $C_name$ and $C_abl$. Variable $C_name$ is used to store and retrieve the names of the signals that are being exchanged, and variable $C_abl$ encodes the states of the employed communication protocol. The sending state machine sets the value of $C_abl$ to 1 to indicate that it wants to communicate. The receiving state machine indicates that it is also able to communicate by setting the value of $C_abl$ to 2. If both state machines are able to communicate, the sending state machine can complete the communication process by setting $C_abl$ to 3. It may also choose to cancel the communication by setting $C_abl$ to 0. The receiving state machine acknowledges successful completion of the communication process by setting $C_abl$ to 0.

#### 3.5.1.7 Making all Signal Names Equal

To keep the transformation that adds the CABP as simple as possible, our implementation of the CABP takes signals with a fixed name as input, transfers them over a lossy channel, and delivers them at the receiving end. Before this instance of the CABP can be used to substitute an asynchronous, lossless, unidirectional channel, the signal names that are sent over this channel have to be changed into this fixed name. Transformation $T_{arg}$ adapts signals such that their name is changed into this fixed name and the name of the original signal is sent as an argument of the resulting signal. For example, the statement **send** *Block*() **to** *O* is replaced by the statement **send** *Signal*("*Block*") **to** *O*.

#### 3.5.1.8 Replacing a Bidirectional Channel by two Unidirectional Channels

Our implementation of the CABP can only substitute asynchronous, lossless, unidirectional channels. In some cases, therefore, a transformation is needed that replaces communication over a bidirectional channel by communication over two unidirectional channels before transformation $T_{ll}$ can be applied. Transformation $T_{uni}$ performs this task.

#### 3.5.1.9 Exclusive Channels for Pairs of State Machines

Transformation $T_{merge}$ cannot merge objects if multiple state machines that are part of an object communicate via the same port. To modify models that do not adhere to this condition, we implemented a transformation $T_{ex}$ that replaces a channel between a pair of objects with a number of identical channels. For each pair of communicating state machines that are part of the two objects, a channel is introduced.

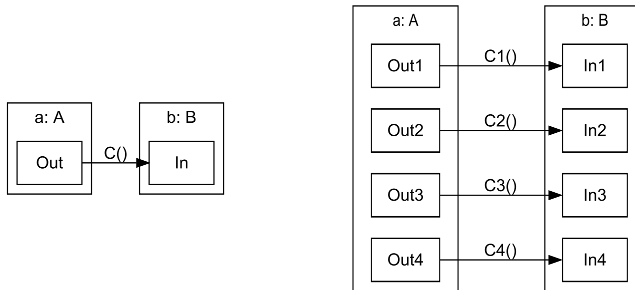



Figure 3.14: Communication before and after adding exclusive channels

On the left of Figure 3.14, two objects are shown that communicate over a single channel. Both objects contain two state machines (which are not shown in this type of diagram) that communicate over this channel. After applying transformation $T_{ex}$, the channel is replaced by four channels, as shown on the right of Figure 3.14.

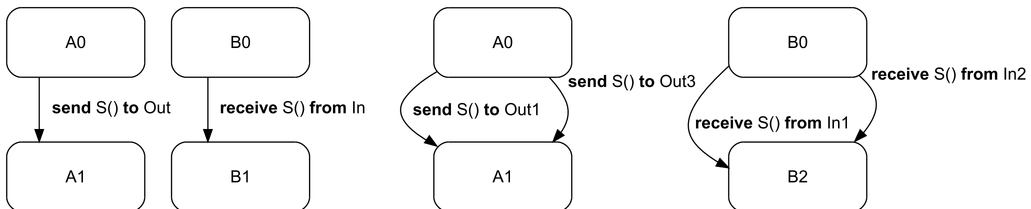



Figure 3.15: State machines before and after adding exclusive channels

As a part of the process of replacing a channel by multiple channels, transformation $T_{ex}$ modifies the state machines that communicate over these channels. Before transformation, both state machines that are part of object *a* send signals via port *Out*, and both state machines that are part of object *b* receive signals via port *In*. After transformation, one of the state machine of object *a* sends signals via ports *Out1* and *Out3*, and the other uses ports *Out2* and *Out4*. The receiving state machines are modified in a similar fashion. Figure 3.15 shows parts of one of the state machine of object *a* and parts of one of the state machines of object *b* before and after transformation. The names of the states of these partial state machines correspond to the names of the classes they belong to. The

situation before transformation is illustrated on the left of the figure, and the situation after transformation is shown on the right.

#### 3.5.1.10 Reducing the Number of Channels

When two objects are connected by more than one channel, these channels can be merged into one if they have the same type and directionality, and support the same argument types. Therefore, we implemented a transformation $T_{mc}$ that merges multiple channels between a pair of objects into one channel. Merging channels is a way of optimizing models because it can be used to reduce the number of instances of the CABP that need to be added.

#### 3.5.1.11 Cloning Classes

Many of the transformations described above use two auxiliary transformations. One of these transformations takes a model and a channel as input, and clones the classes of the objects that communicate over this channel. After applying this transformation, the objects that communicate over the channel are instances of the new cloned classes, and all remaining instances of the original classes remain unchanged. This transformation ensures that all transformations that alter objects that communicate over a certain channel only affect these particular objects.

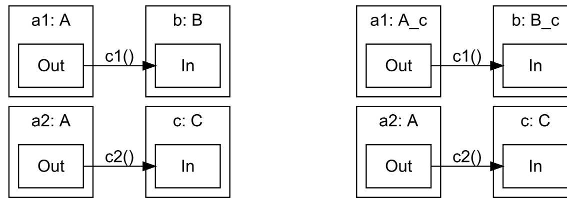



Figure 3.16: A model before and after cloning classes

Figure 3.16 shows the communication diagram of a model before and after applying this transformation. After transformation, the classes of the objects communicating over channel *c1* are cloned. In the resulting model, object *a1* is an instance of class $A_c$ and object $b$ an instance of class $B_c$.

#### 3.5.1.12 Removing Unused Classes

The second auxiliary transformation used by the transformations described above removes all uninstantiated classes from a model. The model depicted on the right of Figure 3.16, for example, no longer contains an instance of class $B$, which means that this class can be removed without affecting the system specified by the model.

### 3.5.2 Exogenous Transformations

Each of the following exogenous model transformations takes an SLCO model as input and produces a model or an implementation in one of the target languages. Because of the gaps described in Section 3.4, the SLCO model provided as input must be specified using a subset of SLCO that matches the capabilities of the target language and platform.

Any model that is specified using constructs that have no direct counterparts in the target language must first be refined using the transformations described in Section 3.5.1.

#### 3.5.2.1 Transforming SLCO to POOSL

```
1  ...
2  process class P()
3    instance variables m: Integer, PSendRecs: String
4    communication channels In1, In2, InOut
5    message interface InOut?T(String); In2?Q(Integer); ...
6    initial method call P_initial()()
7    instance methods
8      P_initial()() | |
9        m := 0; par Rec1_Rec1()() and Rec2_Rec2()() and SendRec_SendRec()() rap
10     .
11     Rec1_Rec1()() | var_1: Boolean |
12       In1?P(var_1 | var_1 = false); Rec1_Rec1()()
13     .
14     SendRec_SendRec()() | |
15       [m = 6] skip; InOut!S("a"); InOut?T(PSendRecs); SendRec_SendRec()()
16     .
17 ...
18     Com_Com0()() | |
19       sel delay(5) or Out1!P(true); Com_Com1()() les
20     .
21 ...
22 behaviour specification
23   (p: P[c3/InOut, c1/In1, c2/In2] || q: Q[c1/Out1, c2/Out2, c3/InOut])
24   \ {c1, c2, c3}
25 .
```

Listing 3.2: Part of a POOSL model

Because of the gaps between SLCO and POOSL, the transformation from SLCO to POOSL is restricted to models that contain only synchronous channels. An example of the output of this transformation is shown in Listing 3.2. This fragment of a POOSL model is the result of applying the transformation to a slightly modified version of the model of Figures 3.5, 3.6, and 3.7. The model has been modified by replacing all asynchronous channels by synchronous channels.

The transformation transforms each SLCO class to a POOSL process class. Lines 2 to 16 of Listing 3.2 show the process class that represents the SLCO class $P$. The state machines of each class are transformed to a number of process methods, one for each state. The method in lines 11 to 13 represents the state *Rec1*, the method in lines 14 to 16 represents the state *SendRec*, and the method in lines 18 to 20 represents the state *Com0*. Besides this, an additional process method is generated for each class, which calls all process methods representing the initial states of the state machines of the class. These additional process methods also initialize the variables of the classes and state machines to their initial values, if applicable. Lines 8 to 10 show the process method that initializes the variable $m$ of class $P$ and calls the process methods that represent the initial states of its state machines as part of a parallel composition.

Since each SLCO class corresponds to exactly one POOSL process class, it is clear that the global variables of an SLCO class can be represented by variables of the corresponding POOSL class. An SLCO state machine, however, is represented by multiple process

methods, and a group of process methods within a process class cannot share variables that are inaccessible by other process methods of that class. For this reason, the local variables of SLCO state machines have to be represented by variables of process classes too. These variables can no longer be considered to be local to a certain state machine, since all the process methods of a class can access them, even those representing other state machines of the same class. Line 3 shows the declaration of the POOSL variables representing the global SLCO variable $m$ and the local SLCO variable $s$ of state machine *SendRec*.

States with a single outgoing transition are translated to process methods containing a single sequence of statements. This sequence of statements represents the outgoing transition. Line 12 shows the sequence of statements representing the outgoing transition of state *Rec1*, and line 15 shows the sequence of statements representing the outgoing transition of state *SendRec*.

States with multiple outgoing transitions are translated to process methods containing a select statement. Each of the alternatives of such a select statement represents one of the outgoing transitions. Line 19 shows the select statement representing the outgoing transitions of state *Com0*. The semantics of the POOSL select statement is as follows. Each of the alternatives of a select statement is a sequence of statements. If one or more of these sequences starts with a statement that is enabled, one of these sequences is chosen non-deterministically and executed. If none of the statements are enabled, the select statement is blocked until one of the statements it contains becomes enabled.

As mentioned above, each transition is transformed to a sequence of statements. If a transition leads to an ordinary state, the sequence of statements representing this transition ends with a process method call. This process method call calls the method representing the target state of the transition. Line 12 shows the sequence of statements that represents the transition from state *Rec1* to itself. If a transition leads to a final state, this process method call is omitted. The first alternative of the select statement on line 19 shows an example of this situation. After 5 ms, the method representing state *Com0* terminates.

POOSL does not offer the form of conditional signal reception that uses expressions as arguments to limit the types of signals that can be received. For this reason, auxiliary variables are introduced to mimic this form of conditional signal reception. Lines 11 and 12 show how variable *var_1* is used to do so.

Lines 23 and 24 show that objects $p$ and $q$ are declared to be instances of classes $P$ and $Q$, and that channels *c1*, *c2*, and *c3* are connected to the ports of these objects.

#### 3.5.2.2 Transforming SLCO to NQC

The transformation from SLCO to NQC transforms SLCO models to NQC implementations, provided that the following two conditions hold. First, all objects describing the behavior of the controllers must communicate with each other via asynchronous, lossy channels. Second, all communication between these objects and the object(s) representing the hardware environment takes place over synchronous channels.

Figure 3.17 shows the state machine of object *ar* that is part of the implementation of the CABP introduced in Section 3.5.1.2 after merging it with objects *a* and *sender*. Listing 3.3 shows a fragment of NQC code that is the result of applying the transformation from SLCO to NQC to a model containing the merged objects.

Every state machine that describes the behavior of an object in an SLCO model is transformed into an NQC task. Lines 2 to 23 show the task that represents the state machine of Figure 3.17. Line 3 shows the declaration and initialization of local variable $b$

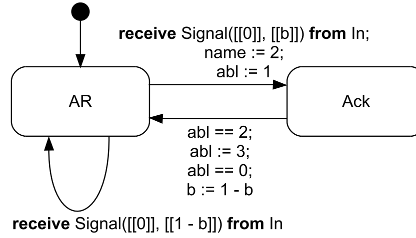



Figure 3.17: Part of the Concurrent Alternating Bit Protocol

```
1  ...
2  task AR() {
3    int b = 0; int temp;
4    AR:
5      temp = Message();
6      if (
7        temp != 0 && ((temp & 112) / 16) == 0 && ((temp & 128) / 128) == b
8      ) {
9        name = 2; abl = 1; goto Ack;
10     }
11     temp = Message();
12     if (
13       temp != 0 && ((temp & 112) / 16) == 0 && ((temp & 128) / 128) == (1 - b)
14     ) {
15       goto AR;
16     }
17   goto AR;
18   Ack:
19     if (abl == 2) {
20       abl = 3; /* skip */; until (abl == 0); b = 1 - b; goto AR;
21     }
22   goto Ack;
23 }
```

Listing 3.3: Fragment of NQC code

and auxiliary variable *temp*. The purpose of auxiliary variable *temp* is explained below. Global variables of SLCO objects, such as *abl* and *name* in Figure 3.17, are represented by global variables of NQC programs, and are not shown in Listing 3.3.

Each state is represented by a label, followed by sequences of statements that represent the outgoing transitions of the state, and a goto statement that jumps back to the label. Lines 4 to 17 show the labeled sequence of statements representing state *AR*, and lines 18 to 22 represent state *Ack*.

If the statements of an SLCO transition start with a statement that might block execution, this statement is translated to an NQC if statement and the remaining SLCO statements are translated to NQC statements that form the body of this if statement. In case the first statement is a signal reception statement, the if statement is preceded by a call to the API function *Message*. The purpose of this API call is explained below. Lines 5 to 10, 11 to 16, and 19 to 21 represent the transitions of the state machine in Figure 3.17. The remaining statements that are part of an SLCO transition are translated as follows. Assignments in SLCO are translated to assignments in NQC, as shown in lines 9 and 20. Expressions and signal receptions are translated to until statements, as

shown in line 20. Statements that send signals are not shown in this example. They are translated to calls to the API function *SendMessage*, using the encoding explained below.

An RCX controller can only send and receive integer values over its infrared port. Since signals in SLCO consist of a name and possibly a number of arguments, they have to be encoded such that they can be represented as a single integer before they can be sent, and they have to be decoded after being received. Lines 5 and 11 show that first the auxiliary variable *temp* is used to store the value of the integer that has been received last. Then, this integer is decoded as shown in lines 7 and 13. The encoding of a signal, which is not shown in this example, is done using a similar procedure. The actual sending and receiving of integer values is done via API calls to the functions *Message* and *SendMessage*, as mentioned above.

In addition to an SLCO model, the transformation from SLCO to NQC also takes a mapping from elements in the SLCO model to concepts of NQC as input. Listing 3.4 shows a part of the mapping for the model described in Section 3.6. For each class that needs to be transformed to an NQC program, this mapping defines how some of the signals of an SLCO model correspond to information received from sensors and commands sent to motors. Each signal that is not mentioned in this mapping is assumed to be part of the communication between controllers via infrared and is translated accordingly, as described above.

```
1  class Middle
2    port2motor
3      Motor -> OutB 7
4    port2sensor
5      Sensor -> Sensor2 Light
6    signal2motor
7      Motor Left -> On Reverse
8      Motor Right -> On Forward
9      Motor Off -> Float
10   signal2sensor
11     Sensor Block -> Below -10
12     Sensor BlockPassed -> Above -2
```

Listing 3.4: Part of a mapping from SLCO to NQC

In line 3, port *Motor* of an SLCO model is mapped to output port *OutB* of a Lego Mindstorms controller. The speed of the motor is set to 7. In line 5, SLCO port *Sensor* is mapped to input port *Sensor2* of a Lego Mindstorms controller, which is connected to a light sensor. Signals named *Left*, *Right*, and *Middle* sent over port *Motor* are mapped to commands for a motor in lines 7 to 9. Lines 11 and 12 show how signals named *Block* and *BlockPassed* are mapped to information received from sensors. In line 11, for example, receiving a signal named *Block* is mapped to the event that the value transmitted by the sensor connected to port *Sensor2* drops below $-10$.

#### 3.5.2.3 Transforming SLCO to Promela

The transformation from SLCO to Promela transforms SLCO models containing only synchronous channels and asynchronous, lossless channels to Promela models. Listing 3.5 shows a fragment of a Promela model that is the result of applying this transformation to a slightly modified version of the SLCO model described in Figures 3.5, 3.6, and 3.7. Because Promela does not support the more general form of conditional signal reception, the

condition $m >= 0$ is removed from the signal reception in the SLCO model. Additionally, the asynchronous, lossy channel is replaced by an asynchronous, lossless channel.

```
1  mtype {S, P, Q, T, a}
2  int p_m = 0
3  chan c1_q2p = [1] of {mtype, bool}
4  chan c3_1_q2p = [0] of {mtype, mtype}
5  chan c3_2_q2p = [0] of {mtype, mtype}
6  ...
7  active [1] proctype p_Rec1() {
8    Label_Rec1: {
9      if :: {c1_q2p?P,eval(false); goto Label_Rec1} fi
10   }
11 }
12
13 active [1] proctype p_Rec2() {
14   Label_Rec2: {
15     if :: {c2_q2p?Q,p_m; p_m = p_m + 1; goto Label_Rec2} fi
16   }
17 }
18 ...
19 active [1] proctype q_Com() {
20   mtype q_s;
21   Label_Com0: {
22     if :: {skip; goto Label_Com2}
23        :: {c1_q2p!P,true; goto Label_Com1}
24     fi
25   }
26   Label_Com1: {
27     if ::
28       {c2_q2p!Q,5; c3_1_p2q?S,q_s; c3_2_q2p!T,q_s; goto Label_Com2}
29     fi
30   }
31   Label_Com2: skip
32 }
```

Listing 3.5: Part of a Promela model

For each object in an SLCO model, the state machines that describe its behavior are transformed to Promela processes. Lines 7 to 11 in Listing 3.5 show the process that represents state machine *Rec1* of object $p$, lines 13 to 17 show the process that represents state machine *Rec2* of object $p$, and lines 19 to 32 show the process that represents state machine *Com* of object $q$.

Channels between objects in SLCO are transformed to channels between processes in Promela. Line 3 shows the declaration of the Promela channel representing the SLCO channel named *c1*. The 1 between square brackets indicates that this channel is asynchronous. The declaration specifies that this channel is suited for messages consisting of two parts: a symbolic name (mtype) and a Boolean value. Lines 4 and 5 show the declaration of two channels representing the SLCO channel named *c3*. Although Promela offers bidirectional channels, a separate channel is declared for each direction. This is done because the semantics of channels in Promela differs from their semantics in SLCO. In SLCO, an object can only receive signals sent by another object. In Promela, however, a process can retrieve a message from a channel that it has sent over this channel itself. For this reason, we introduce two Promela channels for each bidirectional SLCO channel. Each of the Promela channels is used only to communicate in one direction, which means

that a process uses the channel either to send messages or to receive messages, but not both. The 0 between square brackets indicates that this channel is synchronous.

The local variables of state machines are represented by local variables of processes, as shown in line 20. Global variables of SLCO objects are represented by global variables of the Promela model. They are accessible by all processes in the model. On line 2, global variable $m$ of object $p$ is declared.

Ordinary states are transformed to labeled selection statements, and final states are transformed to labeled skip statements. Lines 21 to 25 represent state *Com0* of object $q$, lines 26 to 30 represent state *Com1*, and line 31 represents state *Com2*.

Every outgoing transition of a state is represented by an alternative of the selection statement that represents this state. Each of these alternatives ends with a goto statement to the label representing the target state of the transition. State *Rec1* of object $p$ has only one outgoing transition, as shown in line 9. State *Com0* of object $q$ has two outgoing transitions. The transition to state *Com2* is shown in line 22, and the transition to state *Com1* is shown in line 23.

The semantics of the selection statement is such that it will non-deterministically execute one of the alternatives for which the first statement is executable, and it will block if none of these statements are executable. The statements of a transition are transformed to Promela statements in a straightforward way. Expressions and assignments in SLCO are translated to equivalent expressions and assignments in Promela, as shown in line 15. A signal reception is transformed to a receive statement, as shown in line 9. The receive statement $c1_q2p?P, \mathbf{eval}(\mathbf{false})$ represents the reception of signals named $P$ with an argument that must be equal to **false**. A Promela receive statement blocks until it is able to receive a message over a channel. A send signal statement is transformed to a send statement in a similar fashion, as shown in lines 23 and 28.

## 3.6 Sequences of Transformations

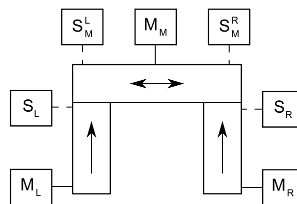



Figure 3.18: Cooperating conveyor belts

To illustrate how the transformations described in Section 3.5 can be applied, we introduce a system of three cooperating conveyor belts, schematically depicted in Figure 3.18. We show how different implementations for controlling this system can be generated by composing a number of transformations into different sequences of transformations. The two vertical rectangles in Figure 3.18 denote conveyor belts that transport items towards another conveyor belt, which is represented by a horizontal rectangle. The arrows in the rectangles denote the directions in which these conveyor belts can transport items. The three belts should cooperate such that items supplied by the vertical belts are dropped onto the horizontal belt, one by one. The horizontal belt should transport each of these items from the right-most belt to the left and those from the left-most belt to the right.

The small rectangles labeled $M_L$, $M_M$, and $M_R$ depict the motors that drive the conveyor belts, and the small rectangles labeled $S_L$, $S_M^L$, $S_M^R$, and $S_R$ depict the sensors that detect the passing items.

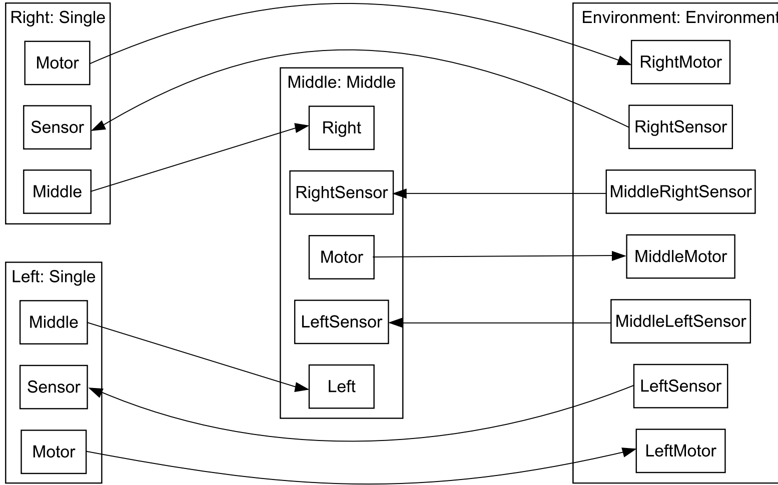



Figure 3.19: Communication between the controllers and the environment

An SLCO model that describes this system consists of four components: two objects (*Left* and *Right*) that model the controllers for the vertical belts, one object (*Middle*) that models the controller for the horizontal belt, and one object (*Environment*) that models the environment. The communication diagram in Figure 3.19 shows how these objects are connected. Among other things, it also shows that the objects *Left* and *Right* are both instances of class *Single*. To increase the readability of the figure, the names of the channels have been omitted.

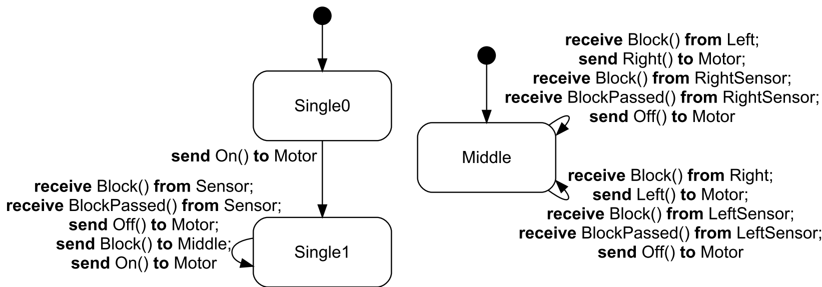



Figure 3.20: Behavior of controllers

Figure 3.20 shows the two state machines that specify the behavior of the controllers of this system. The state machine on the left specifies the behavior of objects *Left* and *Right*, and the state machine on the right specifies the behavior of object *Middle*. The state machines related to the environment are not shown.

In Figure 3.21, three sequences of model transformations leading to three distinct implementations with the same observable behavior are shown. In this figure, SLCO models are represented by the rectangles labeled $M^1_{SLCO}$ to $M^{16}_{SLCO}$, NQC implementations by rectangles labeled $M^1_{NQC}$ to $M^3_{NQC}$, single transformations by labeled, solid arrows, and

subsequences of transformation by labeled, dashed arrows. The starting point of all three sequences, model $M^1_{SLCO}$, describes the intended cooperation in terms of objects *Left*, *Middle*, *Right*, and their hardware environment.

The topmost sequence of transformations in Figure 3.21 transforms the SLCO model $M^1_{SLCO}$ to the NQC implementation $M^2_{NQC}$, which is meant to be deployed on two controllers. First, objects *Left* and *Right* are merged, and the channels that connect these objects to *Middle* are also merged. To be able to distinguish the origin of signals after merging the channels, transformation $T_{ic}$ is applied before merging the channels. Then, synchronous communication is replaced by asynchronous communication, after which lossless communication over a lossy channel is achieved by adding objects that implement the CABP. Because this last transformation adds a number of objects to the model and the resulting model $M^8_{SLCO}$ should only contain as many objects as there are controllers, these objects must be merged with others to reduce the total number of objects again. As mentioned above, objects can only be merged if all pairs of state machines involved in communication communicate over distinct channels. This explains why transformation $T_{ex}$ is applied before transformation $T_{merge}$ is used to reduce the number of objects. In the final transformation step, all string constants are replaced by integer constants, leading to model $M^8_{SLCO}$. Because all semantic gaps and platform gaps have been bridged, transforming this model into implementation $M^2_{NQC}$ using transformation $T_{NQC}$ is straightforward.

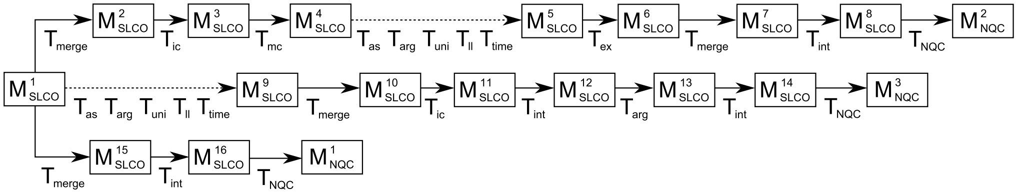



Figure 3.21: Sequences of transformations

The sequence of transformations in the middle of Figure 3.21 leads to an implementation with three controllers. Because the three controllers communicate by broadcasting signals, the origin of signals has to be made explicit using transformation $T_{ic}$. After applying this transformations, the names of the signals have to be changed again, and all string constant have to be replaced by integer constants, to be able to apply transformation $T_{NQC}$. Transformation $T_{NQC}$ encodes the arguments of signals into a single integer, as discussed in Section 3.5.2.2. To keep the number of distinct integer constants as low as possible and achieve an optimal encoding of the signal arguments, transformation $T_{int}$ is applied both before and after transformation $T_{arg}$.

The sequence of transformations in the bottom of Figure 3.21 generates an implementation for a system with only one controller. Therefore, all objects are merged in the first transformation step.

## 3.7 Simplified SLCO

The SLCO metamodel discussed in Section 3.1 and the communication diagrams shown in the rest of this chapter allow a transition to have multiple statements. When a transition is made from one state to another, the statements that are part of this transition are executed one by one. We have already mentioned in Section 3.1 that the execution of

statements related to a given transition can be interleaved by the execution of statements related to a transition of a concurrent state machine. In fact, after executing one of the statements related to a transition, an implicit intermediate state is reached. In some cases, it is convenient to make all these implicit intermediate states explicit. Figure 3.22 show simplified versions of the state machines of Figure 3.7. In these simplified state machines, each transition has at most one statement. Because of this, the state machines have no implicit intermediate states.

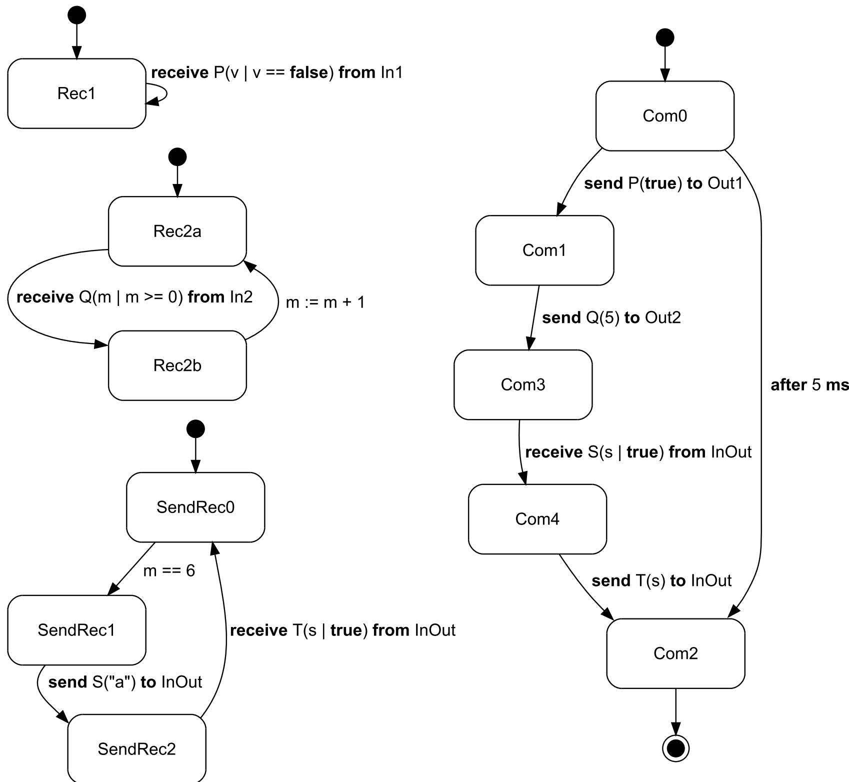



Figure 3.22: State machines of the simplified version of the SLCO model of Figure 3.7

Additionally, the simplified state machines only use the general form of conditional signal reception. To replace the shorthand notation originally used by state machine *Rec1*, an auxiliary variable $v$ is introduced. Furthermore, the expression **true** is added as a condition to the signal reception statements of state machines *SendRec* and *Com*.

The version of SLCO that allows only these simplified state machines is used in Chapter 5, which discusses the process of prototyping the semantics of the language. The formal semantics of SLCO discussed in Appendix B and applied in Chapter 6 is also based on this simplified version of SLCO.

## 3.8 Implementation

In this section, we discuss the implementation of all languages and transformations related to SLCO. The tools used to implement these languages and transformations are described

in more detail in Appendix A. All the grammars, metamodels, and transformations described below are available online[2].

Figure 3.23 gives an overview of the languages and transformations related to SLCO. In this figure, all languages that are based on a metamodel implemented using the Eclipse Modeling Framework (EMF) [106] are depicted as rectangles. The rounded rectangles represent languages that have a textual concrete syntax, and the arrows represent model transformations, parsers, and template-based code generators, which either transform models or convert them from one representation to another.

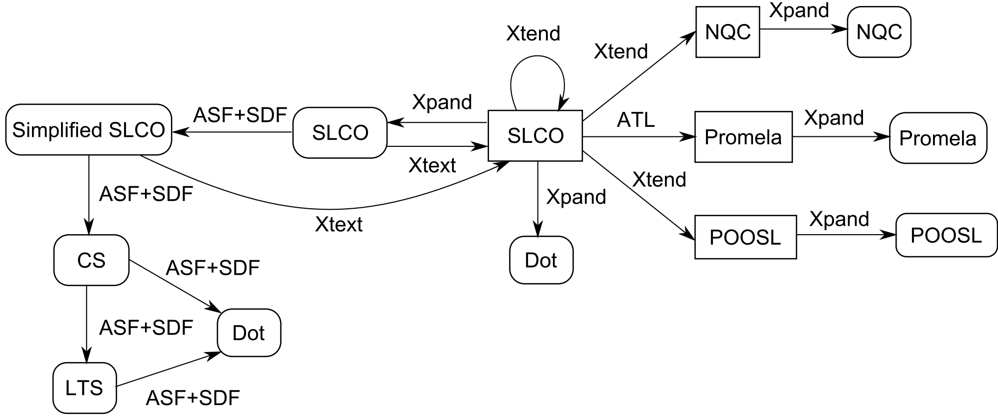



Figure 3.23: Overview of languages and transformations

The metamodels of SLCO, NQC, Promela, and POOSL define the abstract syntax of these languages. They define the concepts offered by the languages and the relations between these concepts. To enable creation of models, EMF offers the automatic generation of a tree-view editor from metamodels.

Unfortunately, creating large models using the standard editor provided by EMF is cumbersome. To ease the process of modeling, we defined a textual syntax for SLCO with Xtext [35], which also provides us with a textual editor for SLCO. Since all other models are automatically generated from SLCO models, there is no need for convenient editors for the other languages involved.

All endogenous transformations that are used to bridge the gaps between SLCO, NQC, Promela, and POOSL are implemented using the Xtend model transformation formalism [50]. In Figure 3.23, these transformations are represented by the arrow that connects the rectangle labeled SLCO to itself. The transformations from SLCO to POOSL and NQC are also implemented using Xtend, and the transformation from SLCO to Promela is implemented using the ATL Transformation Language [62].

The result of each of these transformations is a model that conforms to the corresponding metamodel. These models cannot be used directly in POOSL and Spin or on the Lego Mindstorms platform. Instead, models in textual form are required for simulation, verification, and execution. Therefore, we implemented model-to-text transformations for these tools using Xpand [50]. Additionally, Xpand is used to produce the diagrams that form the graphical syntax of SLCO. Each of the diagrams is in fact a directed graph written in Dot that can be visualized with the Graphviz tool [38].

[2]http://code.google.com/p/simple-language-of-communicating-objects/

The textual languages CS and LTS, their relation to Dot, and the transformations implemented in ASF+SDF [20] are discussed in detail in Chapter 5. The relation between these languages and the simplified version of SLCO discussed in Section 3.7 is also described in Chapter 5. Because every textual model that is expressible in the simplified version of SLCO is also expressible in the regular version of the language, the parser and editor that are generated from the Xtext grammar of SLCO are also suited for the manipulation of simplified textual SLCO models.

Chapter 4

# Exploring the Boundaries of Model Verification

*Traditionally, the state-space explosion problem in model checking is handled by applying abstractions and simplifications to the model that needs to be verified. In this chapter, we propose an approach that applies model-driven software engineering and works the other way around. Instead of making a concrete model more abstract, we propose to refine an abstract model to make it more concrete. Furthermore, we propose to use fine-grained sequences of model transformations to enable model checking of models that are as close to the implementation model as possible. We applied our approach in a case study. The results show that it is possible to validate models that are more concrete when fine-grained sequences of transformations are applied.*

## 4.1 Introduction

Model-driven software engineering (MDSE) is a software engineering paradigm in which models play a central role throughout the entire development process [103]. MDSE combines domain-specific modeling languages (DSMLs) for modeling at a higher level of abstraction and model transformations for the automated generation of various artifacts, such as code, from these models. Our goal is to generate reliable code from models specified using a DSML. To increase the reliability of generated code, formal methods such as verification can be used. Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [26]. An exhaustive state-space search is performed by an automated model checker to determine whether a property holds in a finite-state model of a system. Often, this state space is huge and model checking is no longer a feasible approach for verification. Traditionally, abstractions and simplifications are applied to the model to enable model checking in such cases [23, 25, 113]. We propose an MDSE approach to enable model checking that works the other way around. Instead of starting with a large model and iteratively simplifying it, we start with a small model and iteratively refine it.

In a typical MDSE development process, domain-specific models are iteratively refined using model transformations until a model is acquired with enough details to implement

a system [69]. To increase the reliability of the final system, model checking can be employed. Because of the aforementioned state-space explosion problem, model checking the final system may be infeasible. Instead, we propose to define an additional model transformation that transforms the domain-specific models to models suitable for model checking. Using this model transformation, model checking can be applied on the domain-specific models in every stage of the refinement process. Using this approach, intermediate models close to the implementation can be model checked, while model checking the final system may be infeasible. This chapter thus deals with research question $RQ_2$.

> **$RQ_2$:** *How does the size and complexity of model transformations affect the verifiability of intermediate models produced by sequences of refining model transformations?*

In this chapter, we demonstrate this approach using the Simple Language of Communicating Objects (SLCO), the DSML for modeling systems consisting of concurrent, communicating objects described in Chapter 3. SLCO has an intuitive graphical syntax to model the structure and behavior of a system, and offers constructs such as synchronous communication to make models concise. As discussed in Section 3.5.1, we implemented a number of transformations that can be composed into sequences to transform models specified in SLCO to NQC, a restricted version of C [13]. The semantic properties of NQC and its implementation platform differ from those of SLCO, which means that some constructs that are available in SLCO have no direct counterparts in NQC. To enable transformation from SLCO to the implementation platform, the semantic gaps between the two platforms need to be bridged [8]. Therefore, we added a number of constructs to SLCO and implemented a number of endogenous transformations [79] that can be used to stepwise refine models to align the semantic properties of the DSML with NQC. These transformations replace the constructs in a model that are not offered by NQC by constructs that it does offer, while preserving the observable behavior of the model. A final exogenous transformation transforms the resulting model to executable code.

In Section 3.5.2.3, the transformation from SLCO to Promela is described. Promela is the language of the model checker Spin [55], and this transformation is used to enable model checking of the (intermediate) domain-specific models. Our first experiments showed that verification of models generated by sequences of refining, endogenous transformations using Spin was infeasible due to state-space explosion, even for small source models, as described in Section 4.4.2. We concluded that the change induced on the models by the transformations was too large. In other words, the sequences of transformations were too coarse-grained. Therefore, we split up the coarse-grained sequences of transformations into more fine-grained ones. The impact of the individual transformations of such a fine-grained sequence of transformations on a model is smaller, and the model does not change drastically. This is reflected in the increase of the state space size that is searched by Spin. Using this approach, intermediate models generated by the fine-grained sequences of transformations can be model checked almost all the way up to the models that can be executed because the state space stays within reasonable bounds.

The remainder of this chapter is structured as follows. Our approach to enable model checking of intermediate models on the lowest possible level of abstraction is discussed in Section 4.2. Section 4.3 compares the coarse-grained and fine-grained sequences of transformations that can be used to refine the models created with SLCO. The experiments we conducted are presented in Section 4.4. In Section 4.5, we reflect on our work, and Section 4.6 describes related work. Conclusions and directions for further research are given in Section 4.7.

## 4.2 Approach

Our goal is to generate reliable code from models specified using a DSML. To increase the reliability of generated code, formal methods such as verification can be used. To ensure that the same model is verified and executed, models specified using the DSML should automatically be transformed to models suitable for these purposes. In this way, these models do not have to be created by hand. This enables the use of formal methods without having to create models suitable for that purpose separately. This has the advantage that engineers do not have to learn the syntax and semantics of different languages. Moreover, manual transformation is a slow and error-prone task.

Often, DSMLs and their envisaged implementation platforms have different semantical characteristics. Therefore, the semantic gap between the two formalisms needs to be bridged [8]. We propose to use model transformations to refine DSML models in such a way that the semantic properties of the DSML and the implementation platform are aligned. In this way, the abstract DSML model becomes concrete and transformation from the refined (concrete) model to executable code is merely a syntactical transformation.

To enable verification of DSML models, a transformation from the DSML to a formalism for verification should be implemented. For our experiments, we implemented a transformation to a model checking formalism. Using this transformation, it is possible to verify whether both the abstract and the concrete DSML models fulfill their requirements. From the experiments presented in Section 4.4, we concluded that verification of an abstract model poses no problems. However, verification of a concrete model is infeasible because the verification takes too much time and needs too many resources.

The sequences of transformations used to refine the abstract DSML models produce intermediate models. These models can be transformed to a verification formalism too. By verifying the intermediate models, it is possible to verify models that are more concrete. This approach is schematically depicted in the top half of Figure 4.1. The check marks indicate models that can be verified, whereas the crosses indicate models that cannot be verified. Our experiments showed that it is possible to verify some of the intermediate

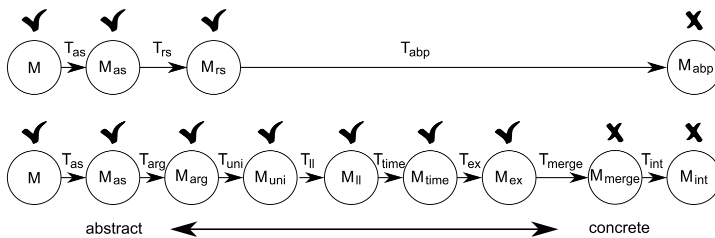



Figure 4.1: Verification of intermediate models

models, but the most concrete model that can be verified contains little implementation details. Because the change induced on the models by the transformations is too large, the intermediate models suffer from state-space explosion after only a few refinement steps. Therefore, we propose to use more fine-grained sequences of transformations to enable verification of more concrete models. This can be achieved by splitting existing transformations into smaller parts. In this way, more intermediate models are generated that can be verified. This approach is schematically depicted in the bottom half of Figure 4.1. Using this approach, it is possible to verify models that are closer to the concrete model. By replacing the transformations $T_{rs}$ and $T_{abp}$ from Figure 4.1 by the

smaller transformations $T_{arg}$, $T_{uni}$, $T_{ll}$, $T_{time}$, $T_{ex}$, $T_{merge}$, and $T_{int}$, for instance, the state space of the intermediate model $M_{ex}$ can be explored, instead of that of the less concrete model $M_{rs}$. The example shown in Figure 4.1 is an illustration of one of the experiments presented in Section 4.4. In different cases, the transformation steps as well as which intermediate models can be verified will vary.

The most concrete model that can be model checked may still not be close enough to the implementation model. An attempt can be made to split the transformations into even smaller parts. If this is not possible anymore, another possibility is to apply the model transformation to part of the model only. Since the refinement, in this case, is applied to a small part of the model, this will most likely result in models that give rise to smaller state spaces. Using partial refinement, the boundaries of what can be verified using model checking can be explored even further.

Using more fine-grained sequences of transformations has some positive side-effects. Since the individual transformations of fine-grained sequences of transformations tend to be smaller than those of course-grained ones, it is easier to locate defects in them. Additionally, these transformations have proven to be more reusable than those used to form course-grained sequences during our experiments. Another advantage of having fine-grained sequences of transformations is that it enables shuffling the order in which the transformations are applied. This order affects the output model, however, and some sequences of transformations may lead to more efficient implementations than others. Furthermore, it may be that not all orderings are allowed because the preconditions of some transformations may be in conflict with the postcondition of others.

## 4.3 Comparison of Transformations

In this section, we compare the transformations used to form coarse-grained and fine-grained sequences of model transformations. The transformations used to form fine-grained sequences, the transformation to Promela, and the transformation to NQC are discussed in Chapter 3. For this reason, we only describe the transformations used to form coarse-grained sequences of transformations in this chapter.

### 4.3.1 Coarse-Grained Sequences of Model Transformations

In Section 4.1, we explained that the characteristics of the platforms differ. To execute SLCO models, the gaps between SLCO and NQC need to be bridged. Therefore, we defined a number of transformations that transform an SLCO model to a refined SLCO model with equivalent observable behavior. Each of these transformations eliminates one of the gaps between the languages and their platforms. An SLCO model that uses synchronous communication only, for example, can be transformed to an equivalent SLCO model that uses asynchronous communication only.

| | **SLCO** | **NQC** |
|---|---|---|
| **(A)synchronous communication** | synchronous and asynchronous | asynchronous |
| **Reliability of communication** | reliable and unreliable | unreliable |

Table 4.1: Language and platform characteristics for the coarse-grained sequences

When starting the development of the sequence of transformations from SLCO to NQC, we identified the gaps between the languages and their platforms shown in Table 4.1.

This analysis showed that, to automatically generate an NQC implementation from an SLCO model, transformations are needed that mimic synchronous communication over an asynchronous channel and that facilitate reliable communication over an unreliable channel. At that point, these gaps seemed to be the most important, and we designed the sequence of transformations such that each of the transformations bridges at most one of the gaps. Two of the following model transformations are implemented to bridge the gaps between SLCO and NQC. The third transformation is used to ensure that models adhere to the precondition of one these two transformations.

#### 4.3.1.1 Synchronized Communication over Asynchronous Channels

The transformation that replaces communication using synchronous signals by communication using asynchronous signals ensures that the behavior of the model is still as desired by adding acknowledgment signals for synchronization. Whenever a signal is sent, the receiving party sends an acknowledgement indicating that the signal has been received. The sending party waits until the acknowledgement has been received. In this way, synchronization is achieved.

#### 4.3.1.2 Lossless Communication over Lossy Channels

Lossless communication over lossy channels is implemented using a variant of the alternating bit protocol (ABP) [11, 12]. This protocol ensures that each signal that is sent, is eventually received, assuming that not all signals get lost. This transformation adds the ABP to a model by adding new state machines implementing the protocol to objects that communicate over a lossy channel. These new state machines communicate with the existing state machines in these objects using shared variables.

#### 4.3.1.3 Exclusive Access to Ports

To ensure that a model meets the precondition of the previous transformation, we use a third transformation. When multiple state machines communicate over the same port, the previous transformation may only be applied if at most one of the state machines sends a message over this port at the same time. The transformation that ensures exclusive access to ports adds a token server to ensure that this is the case. This token server is implemented as an additional state machine that is added to the objects directly. The token server and the existing state machines pass information using shared variables.

### 4.3.2 Fine-Grained Sequences of Model Transformations

Experiments with sequences composed of the model transformations described above showed that these sequences were too coarse-grained. In particular, applying the transformation that adds a variant of the ABP to a model often led to models with a very large state space. Therefore, we reexamined the two languages and identified a number of additional gaps, which are shown in Table 4.2.

When using the coarse-grained sequences of transformations to refine models, the modeler is responsible for creating input models that do not introduce problems concerning the three gaps that are not addressed by the transformations described above. Because these transformations do not introduce objects and cannot be used to reduce the number of objects, the modeler is responsible for creating input models that contain as much objects as can be deployed. The transformations described above also do not introduce data

| | **SLCO** | **NQC** |
|---|---|---|
| **(A)synchronous communication** | synchronous and asynchronous | asynchronous |
| **Reliability of communication** | reliable and unreliable | unreliable |
| **Support for string constants** | yes | no |
| **Connectivity for communication** | point-to-point | broadcast |
| **Number of objects** | $\infty$ | limited |

Table 4.2: Language and platform characteristics for the fine-grained sequences

types that cannot be used in NQC. If the input model does not use these data types, the transformations will result in a deployable model. Because there is no transformation that deals with the problem of identifying the sender of a message that has been broadcasted, only input models with two communicating parties are allowed.

The transformations described in Chapter 3 are replacements of the transformations described above and can be used to compose more fine-grained sequences of transformations. For example, the version of the transformation that ensures lossless communication over a lossy channel described in Section 3.5.1.2 no longer adds state machines to existing objects, but connects these objects to fresh objects that implement the ABP. After adding these objects, another transformation, described in Section 3.5.1.6, must be applied to merge the new and the existing objects, to reduce the total number of objects again and obtain a model that contains as many objects as can be deployed. Also the transformation that ensures exclusive use of ports is replaced by another transformation, which is described in Section 3.5.1.9. The transformation that adds acknowledgements for synchronization, however, is not replaced. This transformation has been expanded to improve its applicability, but this extension is irrelevant for the experiments described below.

## 4.4 Experiments

We performed a number of experiments to determine the size of the state space of intermediate models generated by sequences of refining transformations. By transforming intermediate SLCO models to Promela models, we obtain models whose state space can be explored using Spin. For the experiments described below, we configured Spin to explore the state-spaces by means of a depth-first search with a maximum search depth of $1 \cdot 10^8$ transitions and using at most $4 \cdot 10^4$ megabytes of memory. After describing the models that serve as inputs in our experiment, we show that an approach using coarse-grained sequences of transformations quickly leads to models with very large state spaces. Then, we present the results of our experiments using fine-grained sequences. Finally, we discuss how applying transformations to a part of the applicable model elements only can also be used to explore the state space of less abstract versions of models.

### 4.4.1 Cases

We apply the refining transformations described in Section 4.3 to three different models. The first model consists of one object that repeatedly sends signals via its port (the producer) and one object that is always able to receive signals via its port (the consumer). A synchronous channel connects the ports of the producer to the ports of the consumer. The communication and behavior diagram of this model are shown in Figure 4.2. In a number of steps, channel $c$ is replaced by asynchronous, lossy channels.

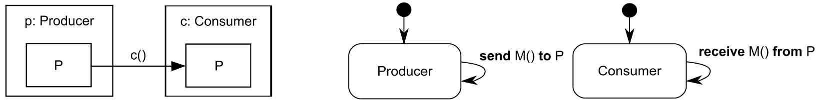



Figure 4.2: A producer and a consumer

The second model describes the behavior of a system consisting of three interoperating conveyor belts. It is a variant of the model described in Section 3.6 that is obtained by merging the objects *Left* and *Right*, and subsequently merging the channels that connect them to object *Middle*. The communication diagram of this model is shown in Figure 4.3, and the behavior diagram is shown in Figure 4.4. The names of the channels are omitted from the communication diagram to increase its readability. The state machine on the right of Figure 4.4 specifies the behavior of object *Middle*, and the behavior of object $L_R$ is specified using two instances of the state machine shown on the left of the figure. The third component, which models the environment of the system, is not described here. This model is transformed by replacing the synchronous channel that connects object $L_R$ and object *Middle* by asynchronous, lossy channels.

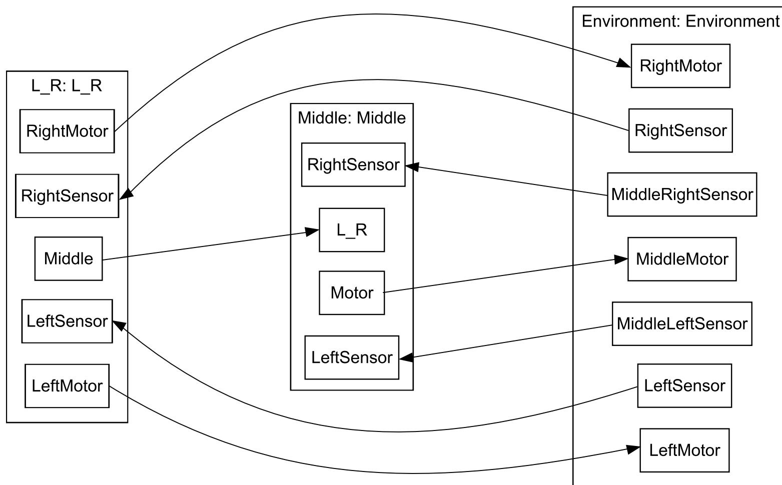



Figure 4.3: Communication diagram of the second model

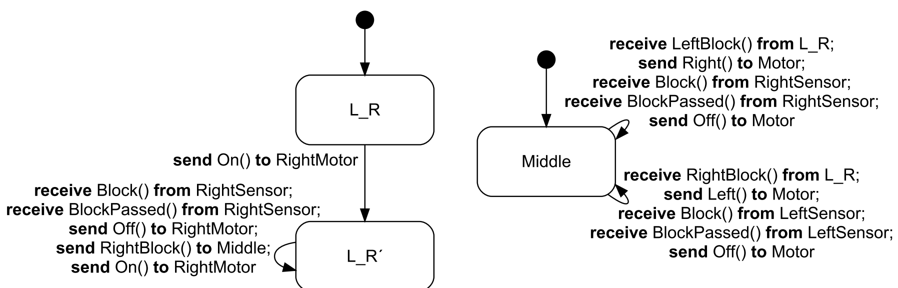



Figure 4.4: State machines of the second model

The third model consists of two objects that repeatedly send signals via their ports (the producers) and one object that is always able to receive signals via two ports (the consumer). Two synchronous channels connect the ports of each of the producers to the ports of the consumer. Figure 4.5 shows the communication and behavior diagram of this model. For the experiments, both channels are replaced by asynchronous, lossy channels.

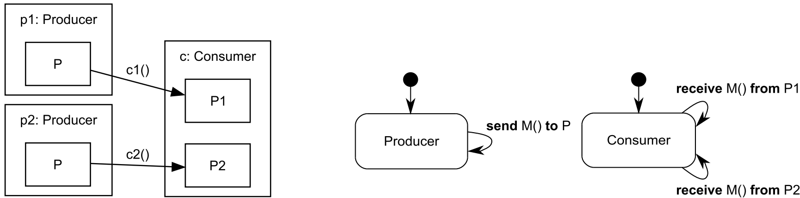



Figure 4.5: Two producers and a consumer

For the experiments described in this section, we used a slightly modified version of SLCO. In this version of the language, the solid black dots represent the initial states, whereas in the current version of SLCO described in Chapter 3, they do not. Instead, the dots and their outgoing arrows are currently only used to indicate the initial states. For the model shown in Figure 4.2, for example, the fact that the black dots represent the initial states themselves means that both the producer and the consumer consist of two states. This is reflected by the tables in the following section. The reason for this difference between the older version of SLCO and the current version is discussed in Chapter 7.

### 4.4.2 Results

Applying a coarse-grained sequence of transformations to the model of the producer and consumer leads to the state space sizes shown in Table 4.3. The table shows that replacing synchronous communication by asynchronous communication approximately doubles the size of the state space. Adding a number of state machines that implement the ABP to each of the two objects, however, leads to a significant increase of the size of the state space. Although the resulting state space of the most concrete model is much larger than the one of the intermediate model, it is still small enough for verification given the aforementioned configuration of Spin.

| **Model** | **# States** | **# Transitions** |
|---|---:|---:|
| Original | 4 | 6 |
| Asynchronous signals | 8 | 11 |
| Lossless communication | 76 066 432 | 542 196 960 |

Table 4.3: Coarse-grained sequence applied to the producer and consumer

To further illustrate the effects of coarse-grained sequences of transformations on the size of the state space, we applied one to the second model, which is slightly more complex than the first. One of the components in the system of the three conveyor belts consists of two instances of the same state machine. Both instances communicate over the same port, which means that a token server must be added when refining this model, before the transformation that adds the ABP can be employed. Table 4.4 shows that adding a token server leads to a state space that can still be model checked. The final row in the

table indicates that it is impossible to explore the entire state space before the search depth is exceeded or all available memory is used. This shows that the output of this transformation is not suited for model checking, even though the input model is still relatively small.

| Model | # States | # Transitions |
|---|---:|---:|
| Original | 494 | 1 294 |
| Asynchronous signals | 748 | 1 980 |
| Token server | 10 090 | 33 820 |
| Lossless communication | – | – |

Table 4.4: Coarse-grained sequence applied to the interoperating conveyor belts

The results of these experiments led us to implement the versions of the transformations discussed in Section 4.3.2. Table 4.5 shows the effect of a fine-grained sequence of transformations on the size of the state spaces of the intermediate models in the case of the producer and the consumer, and Table 4.6 shows the effect in the case of the three interoperating conveyor belts. The transformations that ensure that all signals have a fixed name, replace bidirectional channels by two unidirectional channels, ensure that each state machine within an object communicates with the ABP over an exclusive channel, and replace strings by integers have no effect on the size of the state space.

| Model | # States | # Transitions |
|---|---:|---:|
| Original | 4 | 6 |
| Asynchronous signals | 8 | 11 |
| Fixed signal names | 8 | 11 |
| Unidirectional channels | 8 | 11 |
| Lossless communication | 114 388 | 596 367 |
| Delays | 1 009 856 | 5 902 673 |
| Merged objects | 83 251 840 | 592 242 910 |
| Integers instead of strings | 83 251 840 | 592 242 910 |

Table 4.5: Fine-grained sequence applied to the producer and consumer

| Model | # States | # Transitions |
|---|---:|---:|
| Original | 494 | 1 294 |
| Asynchronous signals | 748 | 1 980 |
| Fixed signal names | 748 | 1 980 |
| Unidirectional channels | 748 | 1 980 |
| Lossless communication | 19 148 872 | 141 049 260 |
| Delays | 167 466 690 | 1 334 614 400 |
| Exclusive channels | 167 466 690 | 1 334 614 400 |
| Merged objects | – | – |

Table 4.6: Fine-grained sequence applied to the interoperating conveyor belts

In the case of the producer and the consumer, each intermediate model has a state-space that can be explored given the aforementioned configuration of Spin. In the case of the conveyor belts, however, merging objects leads to a state-space that is too large to explore. Even though the most concrete model is still unsuited for model checking,

the fine-grained sequence of transformations made it possible to explore an intermediate model that is more concrete than the ones produced using the coarse-grained sequence.

### 4.4.3 Exploring the Boundaries

In both of the cases mentioned above, only two instances of the ABP are added because communication takes place in two directions between one pair of objects. Table 4.7 shows the results for the model consisting of two producers and one consumer. To achieve lossless communication over a lossy channel in this case, four instances of the ABP have to be added, because communication takes place in two directions between two pairs of objects.

| **Model** | **# States** | **# Transitions** |
|---|---:|---:|
| Original | 8 | 17 |
| Asynchronous signals | 33 | 68 |
| Fixed signal names | 33 | 68 |
| Unidirectional channels | 33 | 68 |
| Lossless communication | – | – |

Table 4.7: Fine-grained sequence applied to two producers and a consumer

Adding four instances of the objects that implement the ABP leads to an explosion of the state space. This makes it very hard to verify properties of this model using state-space exploration. Table 4.8 shows the effect of adding an instance of the ABP to respectively one, two, and three channels in the model of two producers and one consumer, while leaving the other channels untouched.

| **Model** | **# States** | **# Transitions** |
|---|---:|---:|
| Original | 8 | 17 |
| Asynchronous signals | 33 | 68 |
| Fixed signal names | 33 | 68 |
| Unidirectional channels | 33 | 68 |
| one ABP instance | 5 188 | 21 335 |
| two ABP instances | 527 108 | 3 224 435 |
| three ABP instances | 105 715 260 | 879 085 750 |

Table 4.8: Incremental introduction of the ABP

By replacing communication over only a subset of the four channels in the model by communication via the ABP, a model is obtained with a state space that is significantly smaller than the state space corresponding to the model in which communication over all channels is replaced. In this way, verification of a model that resembles the implementation more closely than the original, more abstract, model is possible. The same approach can be used to merge only some of the objects in the model of the interoperating conveyor belts. In general, applying a refining transformation to a part of the applicable elements in the model only can be used to model check intermediate models that resemble the implementation as close as possible, in cases where it is impossible to model check the completely refined model.

## 4.5 Discussion

In Section 4.4, we used the model checker Spin to illustrate the effect of both coarse-grained and fine-grained sequences of transformations on state spaces. However, our approach is not limited to one particular model checker. The refining transformations we implemented take SLCO models as input and produce SLCO models as output. Support for another model checker or a similar tool can be added by implementing a single transformation from SLCO to the formalism supported by that tool.

To clearly show the influence of our refining transformations, we used no additional reduction or abstraction techniques. However, our approach can be combined with such techniques in practical situations. Using one of the standard state vector compression modes offered by Spin [54], for instance, it is possible to explore larger state spaces. Using this compression method and the configuration described in Section 4.4, the state space of the timed version of the model of the three conveyor belts can be explored using approximately $15 \cdot 10^3$ megabytes, instead of $31 \cdot 10^3$ megabytes.

Typically, model checking is used to verify whether a property holds for a model of a system. Because the refining transformations modify the model, properties under investigation may have to change as well. After adding communication via the ABP to a model, for example, there are unfair traces in the state space representing the behavior that all signals are discarded by the lossy channel. To consider only the fair traces, a fairness constraint has to be added to the property.

## 4.6 Related Work

Multiple proposals are presented in literature to enable model checking of huge specifications. Clarke et al. suggest four different abstraction techniques and demonstrate their practicality on a number of examples [25]. Another possibility, applied by Chan et al., is to model check only a part of the system [23]. They also applied simplifications to the model to avoid constructs that could not be handled properly by their model checker. Wing and Vaziri-Farahani enabled quick verification in a case study by applying abstractions to both the model and the verification properties [113]. They state that the choice of what abstractions to apply takes some 'good' judgment. All of the aforementioned approaches work by applying abstraction and simplification to concrete models. Our approach works the other way around; we refine an abstract model to a more refined one. Our approach does not preclude the use of abstractions and simplifications on the (intermediate) models. The B-method [1] is developed as a means to refine abstract specifications into implementations. By fulfilling a number of proof obligations and thus proving that each refinement step is sound, it can be proven that an implementation adheres to the corresponding initial specification. Using the B-method, reliable code is derived starting from one initial specification, whereas our approach focuses on automatically generating reliable code from every possible model that can be described using our DSML.

## 4.7 Conclusions and Future Work

In this chapter, we proposed an approach using model checking to increase the reliability of code generated from models specified in a DSML called SLCO. A model transformation from SLCO to a language suitable for model checking has been defined to enable model checking of domain-specific models. Using this model transformation, model checking can

be applied on the domain-specific models in every stage of the refinement process. This chapter addresses research question $RQ_2$ and investigates how the size and complexity of model transformations affects the verifiability of intermediate models produced by sequences of model transformations. Our experiments show that using fine-grained sequences of transformations enables automatically generating more concrete models that are still suitable for explicit state-space exploration in comparison to coarse-grained sequences of transformations. Furthermore, even more concrete models can be obtained by applying transformations to part of a model only.

We conducted experiments to validate our approach on multiple cases related to SLCO. The results show that it is possible to validate models that are more concrete when fine-grained sequences of transformations are applied. In other words, reducing the size and complexity of the refining model transformations improved the verifiability of the intermediate models produced by such sequences. Additionally, since the transformations used to compose fine-grained sequences tend to be smaller than those of coarse-grained sequences, it is easier to locate defects in them. Another advantage of these transformations is their increased reusability in comparison to the transformations used to form coarse-grained sequences.

As discussed in Section 4.5, reduction techniques such as partial order reduction and state vector compression can be applied to a verification model. Additionally, reduction may be applied to domain-specific models. Models in our DSML consist of state machines, and therefore, algorithms for state machine composition [53] may be applicable. Additional research is needed to assess whether reducing the number of state machines in a model leads to smaller state-spaces.

We consider applying the approach to larger models and more complex sequences of transformation to be an interesting direction for future work. The cases on which we applied our technique are rather small, and so are the sequences of transformations. However, we believe that these small examples already show the advantages of the proposed approach. Although the systems under investigation might be larger and more complex in practise, their size and complexity does not bound the verifiability of intermediate models. Instead, the verifiability of these models is bounded by the practical limitations of the hardware used to perform state-space exploration, which determines when state-space explosion causes problems. Thus, our approach of generating the most concrete models whose state space is still suited for state-space exploration is also applicable on larger and more complex problems.

Model checking is one way of increasing the reliability of systems created in an MDSE process. Another way to do this is using formal correctness proofs. When correctness of model transformations can be formally proven, model checking is no longer required to validate intermediate results. It would then suffice to validate the initial model only. Formally proving model transformations requires that the semantics of source and target language are formally defined. Since a lot of DSMLs have an informal semantics only, the correctness of model transformations related to such DSMLs cannot be proven. Therefore, model checking intermediate models may still be required.

# Chapter 5

# Prototyping the Semantics of a Domain-Specific Modeling Language

*A formal definition of the semantics of a domain-specific modeling language (DSML) is a key prerequisite for the verification of the correctness of models specified using this DSML and of transformations applied to these models. For this reason, we implemented a prototype of the semantics of the Simple Language of Communicating Objects (SLCO). Using this prototype, models specified in SLCO can be transformed into labeled transition systems, which allows us to apply existing tools for visualization and verification to models with little or no further effort. By first implementing this executable prototype, we are able to investigate a number of alternative design decisions before specifying a formal semantics for our DSML. The prototype is implemented using the ASF+SDF Meta-Environment, an IDE for the algebraic specification language ASF+SDF, which offers efficient execution of the transformation as well as the ability to read models and produce LTSs without any additional preprocessing or postprocessing.*

## 5.1 Introduction

In Chapter 3, we introduced the Simple Language of Communicating Objects (SLCO), which provides constructs for specifying systems consisting of objects that operate in parallel and communicate with each other. The transformations from SLCO to NQC, POOSL, and Promela described in Chapter 3 provide only a partial transformational description of the semantics of SLCO, because each of these transformations deals with a subset of SLCO. One of the goals of the work presented in this chapter is to define the operational semantics for the entire language.

Another goal of the work presented in this chapter is to facilitate the development of endogenous model transformations [79] for SLCO and to aid reasoning about their correctness. One way to reason about the correctness of model transformations is to compare or relate models before and after transformation, by comparing or relating the state spaces of these models. If the language used to represent such state spaces

is supported by a toolset that offers state space reduction, rather larger SLCO models can be handled, either for analysis of individual models or for comparison of models. As Promela and Spin do not provide support for these features, we have been motivated to look for an alternative solution.

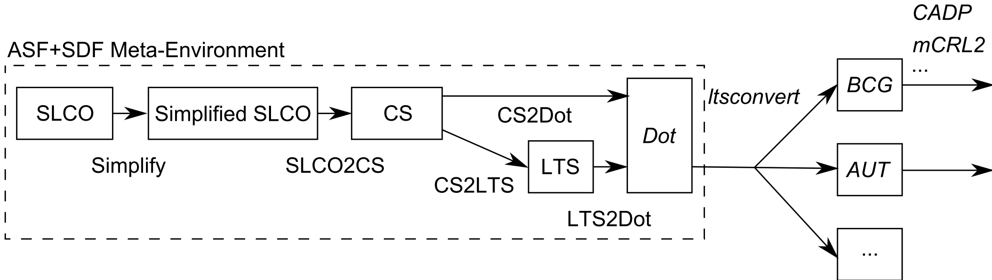



Figure 5.1: Overview of languages and tools

In this chapter, we link SLCO to Dot, a language for the graphical representation of directed graphs [38] that is also used by third-party tools to represent state spaces in the form of labeled transition systems. This link is achieved via a number of transformation tools and intermediate languages, as shown in Figure 5.1. In this figure, the names of existing languages and tools are displayed in an italic typeface. The labeled arrows represent tools, and the rectangles represent languages. Once the translation into Dot has been achieved, several third-party toolsets supporting labeled transition systems are also within reach for manipulation, visualization, and verification of SLCO models. Our languages and transformation tools are defined and implemented in the ASF+SDF Meta-Environment [20].

The process of connecting SLCO to Dot involves two intermediate languages named CS and LTS. The language LTS is a simple language for the representation of labeled transition systems[1]. Due to its simplicity and close resemblance to other representations of labeled transition systems, LTS can easily be linked to existing languages and their supporting toolsets, such as those described in Sections 5.3 and 5.4, but also other languages that may be useful in the future. Together, the tools that transform SLCO models to labeled transition systems represented in LTS form an executable prototype of the semantics of SLCO. This prototype enables investigating alternative design decisions regarding the semantics of the language. By implementing an executable prototype of the semantics of SLCO, we address research question $RQ_3$.

> **$RQ_3$:** *What are the advantages and disadvantages of implementing an executable prototype of the semantics of a domain-specific modeling language using ASF+SDF?*

While objects in an SLCO model are defined separately and interact by communicating over channels, the CS representation of such a model describes its behavior as a whole. The objects are essentially merged into one (big) component, according to the communication as defined in the SLCO model. Thus, CS forms, in a rather natural way, the missing link between the two languages SLCO and LTS. Since the transformation that simplifies SLCO models and the transformation from CS to LTS are straightforward, the transformation

[1] To avoid confusion, we only use the acronym LTS to refer to this representation language in this chapter and do not abbreviate the abstract concept of labeled transition systems itself.

from simplified SLCO to CS essentially forms the core of the prototype semantics of SLCO. The transformation performed by CS2Dot is also straightforward and provides a convenient way to visualize CS representations.

The remainder of this chapter is structured as follows. In Section 5.2, the intermediate languages and the transformation tools shown in Figure 5.1 are introduced. We show how third-party tools can be applied to visualize the state-spaces of SLCO models and explain how these visualizations facilitated the development of SLCO and its accompanying model transformations in Section 5.3. In Section 5.4, applying existing tools for verification of SLCO models is discussed. Section 5.5 addresses the related work, and Section 5.6 concludes the chapter and gives directions for future research.

## 5.2 Prototyping Semantics

Figure 5.1 shows that connecting SLCO to existing tools for verification and visualization involves intermediate languages and transformation tools. Designing this connection and defining its main ingredients required, among others, thorough understanding of the semantics of SLCO and its specification in terms of basic actions. Each of these basic actions represents (a part of) an action performed by a particular object or the result of interaction between objects. The idea behind the intermediate language CS is to explicitly specify these low-level actions, which are implicit in SLCO, and to serve as an underlying language to express the semantics of SLCO. As a result, CS together with the transformation SLCO2CS captures the semantics of SLCO. All languages and transformations described in this section are available for download[2].

### 5.2.1 Languages

The process of transforming an SLCO model into a labeled transition system represented in the language LTS is split into three steps. First, the SLCO model is simplified, as described in Section 3.7. Then, this simplified SLCO model is translated into a list of configurations and steps, represented in the CS language. In this language, we describe the behavior of the entire system, resulting from the communication of the constituting objects, whose behavior is modeled as a set of separate state machines in SLCO. Then, this CS representation is transformed into a list of states and transitions, which form the LTS representation of the input model.

#### 5.2.1.1 CS

The main ingredients in a CS description are configurations and steps. A configuration is a representation of a possible state of the system described by the SLCO model. A configuration can make a step, after which the system reaches another configuration. A step in CS does not necessarily correspond to a single transition in SLCO. Two transitions belonging to two separate objects may lead to a single step in CS if the statements of these transitions send and receive signals over a synchronous channel and thus allow the two objects to communicate synchronously. Conversely, a single transition in SLCO does not necessarily correspond to a single step in CS. For an SLCO transition with a delay statement, several steps (not necessarily executed in sequence) accomplish the behavior specified by the transition.

[2]`http://code.google.com/p/prototyping-slco-semantics/`

```
<
 <p, Rec1, Rec1> <p, Rec2, Rec2a> <p, SendRec, SendRec0> <0, q, Com, Com0>,
 [<<q, Com, s>, ""> <<p, SendRec, s>, ""> <<p, Rec1, v>, false> <<p, m>, 0>],
 [<<c2, q, Out2, p, In2>, > <<c1, q, Out1, p, In1>, >],
 initial
>
```

Listing 5.1: The initial configuration of the running example

**Configurations** Each configuration consists of three mandatory parts and an optional status. The configuration given in Listing 5.1 is the initial configuration of the model consisting of objects $p$ and $q$ shown in Figures 3.5, 3.6, and 3.22. The first part of a configuration specifies the current states of all state machines of all objects in the SLCO model. This part of the configuration is referred to as the active states of a configuration. If a state machine $sm$ of an object $o$ is currently in state $st$, this is specified as $\langle o, sm, st\rangle$ in the active states part of the configuration. This type of active state is referred to as a plain active state. Additionally, a second type of active state exists, which is related to delay statements. These active states are referred to as time-stamped active states. The time-stamped active state $\langle 5, o, sm, st\rangle$ denotes that 5 ms have passed since state machine $sm$ of object $o$ reached state $st$. Furthermore, $\langle \mathbf{passed}, o, sm, st\rangle$ denotes that the maximal amount of time specified by the delay statement of any outgoing transition of state $st$ has passed. The active states part of a configuration consists of a number of plain and time-stamped active states, one for each state machine in the model.

The second part of the configuration, the valuation part, maps variables to values. In the example configuration in Listing 5.1, $\langle\langle q, Com, s\rangle, \text{“”}\rangle$ expresses that the value of local variable $s$ of state machine $Com$ of object $q$ is equal to the empty string. The fact that the value of global variable $m$ of object $p$ is equal to 0 is expressed as $\langle\langle p, m\rangle, 0\rangle$.

The third part of the configuration represents a set of buffers. For each asynchronous channel in the model, one or two buffers are introduced. In case of a bidirectional channel, two buffers are introduced, and in case of a unidirectional channel, one buffer is introduced. The configuration in Listing 5.1 contains two buffers, one for each (unidirectional) channel, and they are both empty. The first buffer corresponds to channel *c2* and the second buffer to channel *c1*. Channel *c3* is synchronous and has no corresponding buffer.

The optional status of this configuration is set to **initial** because all state machine are in their initial state. If all state machines in a configuration are in their final state, the status of this configuration is set to **final**. Finally, the status of all remaining configurations for which there are no steps to other configurations is set to **deadlock**.

**Steps** The dynamics of the system modeled by a set of communicating objects is represented by steps. Each step has a source and a target configuration, and an optional label. A step represents a (basic) action performed by a state machine, the passing of a certain amount of time, or the result of synchronous communication between two objects. Listing 5.2 shows two steps that are part of the CS representation of the SLCO model in Figures 3.5, 3.6, and 3.22. The first step has a label that represents the reception of the asynchronous signal $Q$ by state machine *Rec2*. The second step represents the transition from state *SendRec0* to *SendRec1* of state machine *SendRec*, which is enabled because the expression $m == 6$ holds in the source configuration. Steps that correspond to the evaluation of an expression, such as this one, have no label.

```
<
 <
  <p, Rec1, Rec1> <p, Rec2, Rec2a> <p, SendRec, SendRec0> <q, Com, Com3>,
  [<<q, Com, s>, ""> <<p, SendRec, s>, ""> <<p, Rec1, v>, false> <<p, m>, 0>],
  [<<c2, q, Out2, p, In2>, <Q, 5>> <<c1, q, Out1, p, In1>, <P, true>>]
 >,
 "receive Q(5)",
 <
  <p, Rec1, Rec1> <p, Rec2, Rec2b> <p, SendRec, SendRec0> <q, Com, Com3>,
  [<<q, Com, s>, ""> <<p, SendRec, s>, ""> <<p, Rec1, v>, false> <<p, m>, 5>],
  [<<c2, q, Out2, p, In2>, > <<c1, q, Out1, p, In1>, <P, true>>]
 >
>
<
 <
  <p, Rec1, Rec1> <p, Rec2, Rec2b> <p, SendRec, SendRec0> <q, Com, Com3>,
  [<<q, Com, s>, ""> <<p, SendRec, s>, ""> <<p, Rec1, v>, false> <<p, m>, 6>],
  [<<c2, q, Out2, p, In2>, > <<c1, q, Out1, p, In1>, <P, true>>]
 >,
 <
  <p, Rec1, Rec1> <p, Rec2, Rec2b> <p, SendRec, SendRec1> <q, Com, Com3>,
  [<<q, Com, s>, ""> <<p, SendRec, s>, ""> <<p, Rec1, v>, false> <<p, m>, 6>],
  [<<c2, q, Out2, p, In2>, > <<c1, q, Out1, p, In1>, <P, true>>]
 >
>
```

Listing 5.2: Steps depicting the reception of a signal and the evaluation of an expression

#### 5.2.1.2 LTS

The language LTS is a simple language for representing labeled transition systems as a list of states and a list of transitions. A state can be marked to indicate that it is an initial state, a final state, or a deadlock. Each transition is described as a pair of states, the source and the target state, and an optional label. Listing 5.3 shows the LTS description of a tiny labeled transition system with four states and three transitions. State 0 is declared as an initial state, state 1 is a deadlock, and state 3 is a final state. There are three transitions, one of which has label $a$. The most notable feature of LTS in comparison to existing languages for the description of labeled transition systems is that LTS distinguishes between successful termination and deadlock. Reaching a final state and thus terminating successfully is considered desirable behavior, but reaching a deadlock is not.

```
states
  initial 0
  deadlock 1
  2
  final 3
transitions
  0 1
  0 "a" 2
  2 3
```

Listing 5.3: A small labeled transition system represented in the language LTS

### 5.2.2 Tools for Transformation

Now that the languages used to prototype SLCO are in place, we describe two of the tools that perform transformations related to these languages. The first tool produces CS representations from simplified SLCO models, and the second tool produces labeled transition systems represented in LTS from CS representations. The tools that perform the transformation from LTS to Dot and the transformation from CS to Dot are discussed in Section 5.3, and the simplification of SLCO is discussed in Section 3.7.

The aforementioned tools are implemented in the ASF+SDF Meta-Environment [20], which is described in Appendix A. The main benefits of using the ASF+SDF Meta-Environment for this task are that it offers an IDE for the convenient development of transformations as well as automatic generation of command-line tools. These command-line tools are fast and make efficient use of memory, which is important when generating CS and LTS representations of large state spaces.

#### 5.2.2.1 SLCO2CS

An SLCO model is translated into CS in three phases. First, the initial configuration of the model is constructed. The list of active states of this configuration consists of the initial states of each of the state machines of the objects in the model, the valuation maps all variables to their initial values, and the buffers corresponding to all asynchronous channels are empty. Second, the set of all reachable configurations is generated. This phase is described in more detail below. Third, the list of configurations is traversed to find the configurations containing only active states that are final and those that have no outgoing steps. The configurations containing only final active states are marked as final, and the configurations that have no outgoing steps are marked as deadlocks, unless they are already marked as final.

In the second phase, first all configurations that are reachable from the initial configuration are created, as well as all the steps to these configurations. Then, all configurations that are reachable from these new configurations and the corresponding steps are created, and so on, until no new configuration is found. The configurations that are reachable from a source configuration are computed based on the active states of this source configuration. The ASF+SDF functions discussed next are selected from the set of all functions that together implement the generation of configurations and steps within the second phase of the transformation.

Listing 5.4 shows one of the conditional rewrite rules that implement the function *takeStepsFromConfiguration*. In this rule and all other ASF rules shown in this chapter, variable names start with a dollar sign. Furthermore, variable names that contain a multiplication symbol (`$X*`) represent lists of terms of a sort, variable names that contain a plus symbol (`$X+`) represent non-empty lists of terms of a sort, and variable names that end with a question mark (`$X?`) represent an optional term that can be omitted. The rule in Listing 5.4 shows that the computation of possible steps from a given configuration is split into two parts. First, function *takeTimeStepsFromConfiguration* computes a step that represents the passing of a certain amount of time, if such a step is possible from the given configuration. This computation deals with transitions with delay statements only. Then, function *takeStepsActiveStates* inspects all outgoing transitions of the active states of the source configuration to compute the rest of the reachable configurations.

The ASF rule in Listing 5.5 shows how a step representing the passing of time is computed. The function *getSmallestTimeStepFromConfiguration* inspects all outgoing transitions of the active states of the given configuration and returns a natural number.

```
<$Configuration*0, $Step*0> :=
  takeTimeStepsFromConfiguration($Model, $Configuration),
$ActiveState* := activeStates($Configuration),
<$Configuration*1, $Step*1> :=
  takeStepsActiveStates($Model, $Configuration, $ActiveState*)
====>
takeStepsFromConfiguration($Model, $Configuration) =
<$Configuration*0 $Configuration*1, $Step*0 $Step*1>
```

Listing 5.4: ASF rule that computes reachable configuration and the corresponding steps

This number is computed from the time stamps of time-stamped active states and the outgoing transitions of these active states. By construction, all active states that have an outgoing transition with a delay statement also have a time stamp. For each of the time-stamped active states, function *getSmallestTimeStepFromConfiguration* uses their time stamp to compute the smallest amount of time that must pass for one of the delay statements of the related outgoing transitions to become unblocked. After this amount of time is computed, the resulting configuration and step are constructed by updating the time stamps of all time-stamped active states.

```
canTakeTimeStepFromConfiguration($Model, $Configuration) == true,
$NatCon := getSmallestTimeStepFromConfiguration($Model, $Configuration),
$Configuration0 := updateTimeStamps($Model, $Configuration, $NatCon),
$StrCon := natCon2StrCon($NatCon) || " ms",
$Step := <$Configuration, $StrCon, $Configuration0>
====>
takeTimeStepsFromConfiguration($Model, $Configuration) =
<$Configuration0, $Step>
```

Listing 5.5: ASF rule that computes a step representing the passing of time

Figure 5.2 and Listing 5.6 illustrate the computation performed by the ASF rule in Listing 5.5. In the initial configuration, the leftmost state machine in Figure 5.2 is in state *A0*, and the rightmost state machine is in state *B0*. Both states have a time stamp equal to 0. The smallest amount of time that must pass for one of the delay statements to become unblocked is 1 ms. The first step in Listing 5.6 corresponds to the passing of this amount of time. In the resulting configuration, the smallest amount of time that must pass for one of the remaining delay statements to become unblocked is 2 ms, which is represented by the second step in Listing 5.6. In the resulting configuration, both delay statements of the leftmost state machine are unblocked, which is indicated by the keyword **passed** in the time-stamped active state $\langle \mathbf{passed}, a, A, A0 \rangle$. Finally, for the remaining delay statement to become unblocked, 3 ms must pass, as shown in the third step in Listing 5.6.

After constructing the reachable configurations and the corresponding steps related to time, all other configurations that are reachable from the source configuration are computed by the function *takeStepsActiveStates*, as mentioned above. For each active state in a configuration, the outgoing transitions are inspected. Whether a transition is enabled depends on the valuation of the variables, the contents of the buffers, and the values of the optional time stamps. The valuation of the variables is used to determine whether the expressions of transitions hold, the content of the buffers to determine

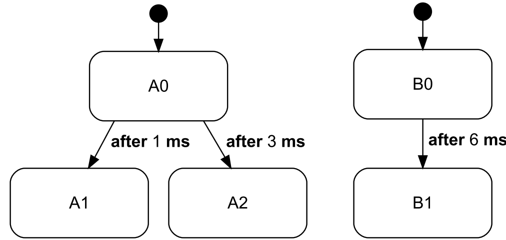



Figure 5.2: Two state machines with delay statements

```
<
 <<0, a, A, A0> <0, b, B, B0>, [], []>,
 "1 ms",
 <<1, a, A, A0> <1, b, B, B0>, [], []>
>
<
 <<1, a, A, A0> <1, b, B, B0>, [], []>,
 "2 ms",
 <<passed, a, A, A0> <3, b, B, B0>, [], []>
>
<
 <<passed, a, A, A0> <3, b, B, B0>, [], []>,
 "3 ms",
 <<passed, a, A, A0> <passed, b, B, B0>, [], []>
>
```

Listing 5.6: Steps representing the passing of time

whether any signal receptions are possible, and the time stamps to determine which delay statements are no longer blocked.

```
$ActiveState0 := getNextState($Model, $ActiveState, $Transition),
$IdCon0 := getObjectId($ActiveState),
$IdCon1 := getStateMachineId($ActiveState),
$AssignmentStatement := getStatement($Transition),
<$Configuration0, $Step0> := processAssignmentStatement(
  $AssignmentStatement, $ActiveState, $ActiveState0, $Configuration,
  $IdCon0, $IdCon1
)
====>
takeStepTransition($Model, $Configuration, $ActiveState, $Transition) =
<$Configuration0, $Step0>
```

Listing 5.7: Computing the configuration that is reached after executing an assignment

Listing 5.7 shows one of the conditional rewrite rules that implements the function *takeStepTransition*. This function is used by the function *takeStepsActiveStates* to construct the reachable configurations and the corresponding steps for each of the enabled outgoing transitions of a certain configuration. The rule in Listing 5.7 applies to transitions with an assignment statement. First, the function *getNextState* is used to compute the active state that results from taking the transition at hand from the original active state. Then, the function *processAssignmentStatement* is applied, which produces

the configuration reachable from the source configuration and the corresponding step. The configuration is an updated version of the configuration provided as input, in which the active state \$*ActiveState* is replaced by \$*ActiveState0* and the valuation is adapted according to the assignment statement. The step consists of the original configuration and the updated configuration.

```
$IdCon2 := $Expression := $AssignmentStatement,
$ConstantExpression :=
  evaluateExpression($Configuration, $Expression, $IdCon0, $IdCon1),
$Value := constantExpression2Value($ConstantExpression),
$Configuration0 :=
  updateActiveState($Configuration, $ActiveState0, $ActiveState1),
$Configuration1 :=
  updateNameValue($Configuration0, $IdCon0, $IdCon1, $IdCon2, $Value),
$StrCon0 := idCon2StrCon($IdCon2),
$StrCon1 := value2StrCon($Value),
$StrCon2 := $StrCon0 || " := " || $StrCon1,
$Step := <$Configuration, $StrCon2, $Configuration1>
====>
processAssignmentStatement(
  $AssignmentStatement, $ActiveState0, $ActiveState1, $Configuration, $IdCon0,
  $IdCon1
) = <$Configuration1, $Step>
```

Listing 5.8: Processing an assignment statement

The ASF rule that implements the function *processAssignmentStatement* is shown in Listing 5.8. The listing shows how the expression that is part of the assignment statement is evaluated using the configuration provided as input. After a new configuration is constructed by updating the active states and the valuation of the variables, a label is generated that reflects the assignment that is carried out.

The details of the rules that handle the other types of statements offered by SLCO differ from those described above, but the general idea is the same for all rules.

#### 5.2.2.2 CS2LTS

The tool CS2LTS translates lists of configurations and steps from CS to LTS, as shown in Figure 5.1. Each configuration is mapped to a unique natural number and an optional status. The status indicates whether a state is an initial state, a deadlock, or a final state, and it is equal to the status of the corresponding configuration. Each step is transformed to a pair of natural numbers representing its configurations, possibly decorated by an optional label. The label of a transition is equal to the label of the corresponding step.

## 5.3 Visualization

We use Dot, which is described in Appendix A, to visualize the state spaces of SLCO models by translating LTS representations to graphs in the format of the Dot language. State spaces represented in LTS are transformed to Dot graphs by translating all states to nodes and all transitions to edges, and specifying how initial states, final states, and normal states should be visualized.

Figure 5.3 shows the labeled transition system that represents the state space of the model in Figure 3.5, 3.6, and 3.22, visualized using Dot. In the visual representation, the

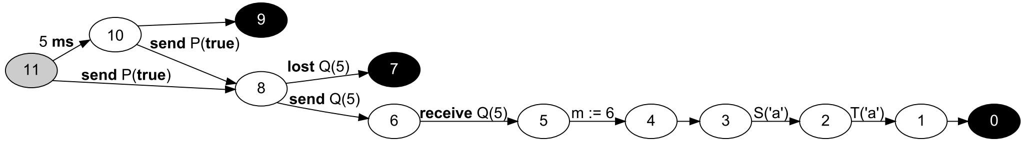



Figure 5.3: A state space visualized using Dot

transition labels are placed above the corresponding transition. The unlabeled transitions in the state space of Figure 5.3 correspond to transitions with expressions in the SLCO model whose behavior is described by the state space.

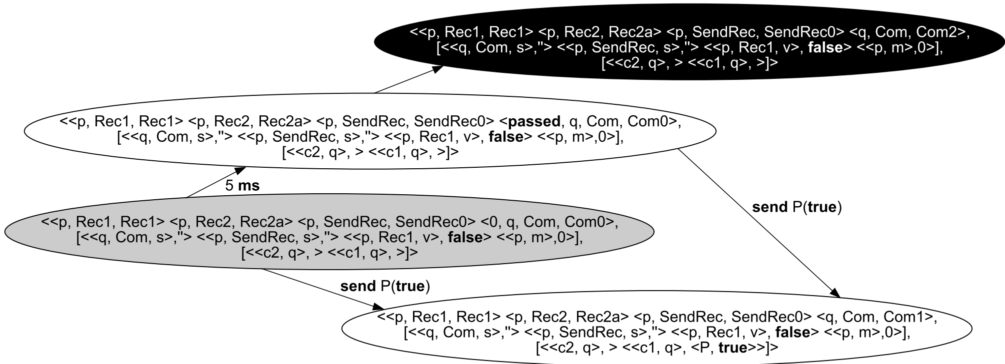



Figure 5.4: A more detailed visualization for debugging

CS representations of configurations and steps, such as the ones in Listings 5.2 and 5.6, list all reachable configurations and steps for a given SLCO model. Therefore, design decisions can be evaluated by inspecting the CS representations of a number of models while developing the executable prototype of the semantics of SLCO. However, visual representations of the same information, such as the state space in Figure 5.3, provide a more convenient way of checking whether the semantics implemented by means of the various transformation tools coincides with the intended semantics. Using the visual representation, it is often easier to spot unintended behavior.

Once unintended behavior is encountered, more information about the configurations can help to locate and repair the parts of the tools that cause this behavior. The desire for more detailed information lead to the implementation of the tool CS2Dot, which translates CS representations directly to more elaborate Dot graphs. In Figure 5.4, the four leftmost states of the state space in Figure 5.3 are shown in this elaborate graphical representation, along with a number of the related transitions.

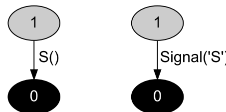



Figure 5.5: State spaces before and after refinement

The various textual and graphical representations of the behavior of SLCO models described above have also been used during the development of model transformations. By

generating the state space of both the source and target model for a given transformation and comparing these state spaces, the effect of the transformation can be studied. For instance, Figure 5.5 shows the state space of a small SLCO model before and after applying transformation $T_{arg}$, which is described in Section 3.5.1.7. The figure clearly shows that the transformation has no unwanted side effects. Because of its complexity, we used this approach to develop transformation $T_{as}^{G}$ described in Section 3.5.1.1.

## 5.4 Verification

For small state spaces, like the one in Figure 5.3, it is easy to verify properties manually by inspecting the graph. In case of larger state spaces, reduction techniques can be applied to first reduce the labeled transition system that represents the state space before verifying properties manually. The tool ltsconvert, which is part of the mCRL2 toolset [49], takes a labeled transition system in various formats as input and converts it to an equivalent labeled transition system in another format. One of the formats that ltsconvert is able to process is the Dot format. The tool is also capable of reducing labeled transition systems by means of an equivalence relation. It supports several of these equivalence relations.

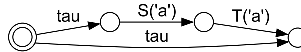



Figure 5.6: A reduced LTS visualized using Dot

Figure 5.6 shows the state space obtained after reduction has been applied to the state space in Figure 5.3. In this case, branching bisimilarity [47] was used as equivalence relation. The labeled transition system has been reduced by, first, turning all labels except $S$('a') and $T$('a') to internal, unobservable labels and then removing all redundant states and transitions using ltsconvert. A similar procedure can be applied for the other reductions that are supported by this tool.

A number of the transformations described in Section 3.5.1 transform SLCO models into other SLCO models with equivalent observable behavior. By producing a labeled transition system representing the state space of the input and output model of such a transformation and reducing both labeled transition systems using the technique described above, the correctness of this transformation for the given input model can be verified by comparing the reduced labeled transition systems. If both reductions lead to the same labeled transition system, the transformation has indeed preserved the observable behavior. We used this approach to assess the correctness of the transformations described in Sections 3.5.1.1, 3.5.1.2, and 3.5.1.6 by applying these transformations to a number of models and comparing the state spaces of these models before and after transformation.

When labeled transition systems get too large for reduction and manual inspection, other tools can be used for verification. One approach is converting labeled transition systems to the BCG and AUT file formats that are used by the CADP toolset [42] to represent labeled transition systems. The CADP toolset offers tools that take a labeled transition system and a temporal logic property as input and perform on-the-fly verification of the property on the labeled transition system. Alternatively, the previously mentioned mCRL2 toolset can be used for verification too. This toolset includes a tool that can transform labeled transition systems to the proprietary format of the toolset as well as tools that can be used to analyze, simulate, manipulate, and visualize models described using this format. These two example toolsets clearly show the added benefit

of producing labeled transition systems from SLCO models. Transforming models to this common description format makes it possible to verify properties of models using existing tools, without additional effort.

## 5.5 Related Work

Hooman and Van der Zwaag [56] used the interactive theorem prover PVS to define the semantics of a subset of the UML. In this subset, the behavior of objects is specified using state machines that communicate with each other both synchronously and asynchronously. Proving properties of models in this approach is done manually using PVS. This is a complex task that requires expertise in PVS, which can be simplified using certain predefined strategies. A disadvantage of this approach is that is does not offer the reusability of other existing tools that our approach offers. An advantage of this approach is that it does not suffer from the state-space explosion problem, because the complete state space of models does not have to be generated for property verification.

Di Ruscio [33] et al. define the semantics of a DSML for the development of telephony services using Abstract State Machines (ASMs). Because ASMs can be executed, this definition can be used to simulate models specified in their DSML. The approach is meant for the specification of the behavioral semantics of the DSML only and does not offer verification of models. Proving properties for all models in general or any specific model is not investigated. In theory, however, properties of models could be verified in the domain of ASMs.

Sadilek and Wachsmuth [99] propose a technique for defining the semantics of DSMLs that uses model instances as configurations and QVT relations to define steps between configurations. Configurations, representing model instances, can be visualized using the same editors used to create models. By reusing the existing editors, visual interpreters and visual debuggers can be created with relatively little effort. Although this technique is suited for simulation of models, it is not efficient enough for state-space generation. Because each configuration is represented by a model, a lot of memory is needed to store all possible configurations.

A number of approaches use Maude to specify the operational semantics of DSMLs [97, 98]. Given the operational semantics of a DSML in Maude, other techniques can be applied to verify properties of models specified in such a DSML. Both an LTL model checker [37] and a $\mu$-calculus model checker [112] are available for rewrite systems specified in Maude. Although it is clear that model checking techniques can be implemented in Maude and applied to specifications of the semantics of DSMLs, not all techniques applicable to labeled transition systems that we aim to exploit, such as reduction and visualization, have been implemented in Maude. It might be the case, therefore, that a given technique must first be implemented in Maude before it can be used in combination with a specification of the semantics of a DSML. With our approach, we can connect to various tools and apply existing techniques only by adapting the representation of labeled transition systems, if needed.

## 5.6 Conclusions and Future Work

In this chapter, we addressed research question $RQ_3$ by implementing an executable prototype of the semantics of SLCO using ASF+SDF. We defined the semantics of SLCO by implementing a number of tools that transform SLCO models to representations of

labeled transition systems, which has a number of advantages. First, various existing tools for visualization of state spaces and verification can be reused because labeled transition systems are commonly used as input by such tools. This provides the opportunity to apply these tools to verify and visualize SLCO models, which aids the development of SLCO itself as well as the related model transformations. Furthermore, based on the work described in this chapter, we defined a formal semantics of SLCO, which is described in Appendix B. The prototype of the semantics of SLCO and its implementation as presented here provided a solid foundation for this work, as we got better understanding of the semantics of our DSML and were able to investigate a number of design decisions.

We implemented the transformation tools presented in this chapter using the ASF+SDF Meta-Environment. In this way, we investigated its suitability for the purpose of implementing an executable prototype of the semantics of a DSML. The biggest advantages of using the ASF+SDF Meta-Environment for this implementation are ATerms [21], used for representing terms, and the command-line tools that can be automatically generated. The use of ATerms guarantees efficient use of memory, and the command-line tools offer efficient execution of rewrite rules, without any additional effort during the implementation. Both execution speed and efficient use of memory are important in this case because the state spaces of models represented by labeled transition systems are typically very large. Unfortunately, active development of ASF+SDF has stopped.

As mentioned above, a formal semantics of SLCO has been developed, which is based on the prototype described in this chapter. However, the notion of time is not yet incorporated in the formal semantics of SLCO as presented in Appendix B. Extending the formal semantics by including time is left as future research. Additionally, we consider to apply the approach taken in this chapter to other DSMLs. The approach lends itself well for the creation of state-space generators based on the operational semantics of a given DSML and prototyping the semantics of languages with informal or incompletely defined operational semantics.

# Chapter 6

## Reusability and Correctness of Endogenous Model Transformations

*Correctness of model transformations is a prerequisite for generating correct implementations from models. Given refining model transformations that preserve desirable properties, models can be transformed into correct-by-construction implementations. However, proving that model transformations preserve properties is far from trivial. Therefore, we aim for simple correctness proofs by designing sequences of model transformations for automated code generation that are as fine-grained as possible. Furthermore, we advocate the reuse of model transformations to reduce the number of proofs. For a simple domain-specific modeling language, SLCO, we define a formal framework to reason about the correctness, reusability, and composition of the model transformations used to transform a given model to three target languages: NQC, Promela, and POOSL. The correctness criterion induces that the systems specified by the original model and the resulting model obtained after a proper sequence of transformations have the same observable behavior.*

## 6.1 Introduction

When generating executable code from models by applying (sequences of) model transformations, the properties modeled initially in the source model have to be propagated to the target implementation. In general, the target implementation is more complex, due to added implementation details, which makes its analysis more difficult, more time consuming, and in some cases even impossible, as described in Chapter 4. The source model, however, is relatively small, so its properties can be inspected and validated. To check preservation of properties, one can analyze all or some of the intermediate models, but this means that a large portion of the analysis has to be duplicated. In addition, this procedure has to be repeated for every new model, even if only small changes to the model have been made.

The efficient and general solution to this problem is to prove property preservation per transformation and to localize the proof only on the changes induced by the transformation.

In this chapter, we address research question $RQ_4$ by presenting an approach that provides such a solution.

> **$RQ_4$:** *Can we show that the model transformations that we implemented to refine SLCO models preserve certain desirable properties of such models?*

Our approach is demonstrated on SLCO, the small but non-trivial domain-specific modeling language (DSML) introduced in Chapter 3. SLCO is a DSML for the specification of systems consisting of objects that operate in parallel and communicate with each other. In SLCO, such systems can be specified on various levels of abstraction. From a given SLCO model on a high level of abstraction, different compositions of model transformations are used to generate NQC [13] models for execution on Lego Mindstorms controllers, POOSL [108] models for simulation, and Promela models for formal verification using the model checker SPIN [55]. The transformations from SLCO to these languages are described in Section 3.5.2. The part of SLCO used for the specification of high-level models and the three target languages have different properties, and therefore, several semantic gaps need to be bridged [8], which are described in Section 3.4. Each of these gaps is bridged by one or more model transformations that add implementation details to the original SLCO model, resulting in a refined SLCO model that is closer to one of the target languages. To improve the reusability of these transformations and to deal with only one language for the majority of the correctness proofs, we only use endogenous transformations for the refinement of models, instead of exogenous ones [79]. These endogenous transformations are described in Section 3.5.1. To be able to use endogenous transformations for refinement, we extended SLCO with constructs to specify systems on a lower level of abstraction too.

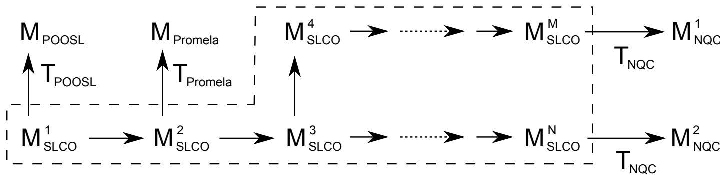



Figure 6.1: Sequences of transformations for three target languages

Figure 6.1 schematically depicts a number of composed model transformations that transform an SLCO model to various target languages. The arrows inside the dashed shape depict endogenous transformations that transform SLCO models into more refined SLCO models. Each of the endogenous transformations leads to a model with observationally equivalent behavior. The arrows across the border of the dashed shape depict exogenous transformations. Because the semantic gaps between SLCO and the target languages are bridged completely by the endogenous transformations, these exogenous transformations are straightforward translations of SLCO constructs into equivalent constructs in the target languages. By developing sequences of endogenous transformations that are as fine-grained as possible, we improve the reusability of the individual transformations within sequences and reduce the number of complicated proofs.

In this chapter, we discuss how the transformations of SLCO models to the three target languages are decomposed into sequences of transformations, and the way these transformations are composed and reused within the sequences. Furthermore, we describe a formal framework for SLCO used to reason about the correctness of model transformations. First, the formal structural operational semantics (SOS) [93] of SLCO is defined, which

generates a labeled transition system (LTS) representation of the dynamics of an SLCO model. The semantics described in this chapter are based on our experience with the executable prototype described in Chapter 5 and describe the behavior of an SLCO model in terms of the behavior of the individual transitions of state machines in the model. Second, for each transformation, a (behavioral) equivalence relation is established between the behavior of an SLCO model before and after transformation. Finally, a proof of the correctness of a transformation is given by showing that the transformation preserves the corresponding equivalence relation.

The benefits of this approach are manifold. The correctness of the aforementioned model transformations is proven in general, not only for particular model instances. The use of SOS and the behavioral equivalence relations allows us to focus only on the part of a model affected by a transformation when reasoning about the correctness of this transformation. Furthermore, the constraints on input models required for some of the transformations, detected earlier during experimental work, can now be formally shown necessary for the correctness of the transformations. This proves that a sequence of transformations is well-composed if each of the intermediate models satisfies the constraints that must hold for the transformation that is applied next.

The remainder of this chapter is structured as follows. In Section 6.2, the model transformations related to SLCO and the way they are reused are described. The correctness criterion and a proof of one of the transformations are given in Section 6.3. This section also describes the formal semantics of SLCO, which is required for the proof. Section 6.4 discusses the related work, and Section 6.5 concludes this chapter.

## 6.2 Model Transformations for SLCO

DMSLs allow designers to reason at a high level of abstraction, and therefore, DSML models do not include many implementation details. The main goal of refining model transformations is to add more details to the model, thus bringing it closer to its implementation. To generate code (such as an NQC executable) from an SLCO model, a number of endogenous model transformations have been designed and implemented. By design, each model transformation transforms only a specific small part of the input model, because small transformations can be easily applied, composed, implemented, and analyzed. We have composed sequences of transformations for several target languages. Our correctness criterion guarantees that every intermediate model, including the last model in the sequence, has the same properties as the source model. Furthermore, the transformations are designed and composed such that the very last SLCO model in the chain contains all implementation details.

### 6.2.1 Reusability of Model Transformations

Table 6.1 lists eleven of thirteen endogenous model transformations that are defined to refine SLCO models. The other two transformations deal with time and are not discussed in this chapter. A detailed discussion of all these transformation is given in Section 3.5.1.

There are two ways in which these transformations can be reused. First, a model transformation can be applied multiple times within the same sequence of transformations, as indicated in the second column of Table 6.1. In practice, the most reused model transformations are the *Clone Classes* transformation, which can be used to clone certain classes, and the *Remove Classes* transformation, which can be used to remove all classes

that have no instances. They ensure that models adhere to the constraints imposed by most of the other transformations and are therefore crucial for the successful composition of transformations. Second, a model transformation can be applied in multiple sequences leading to different target languages, as indicated in the third column. This type of reuse is less common, but supporting other target languages with similar semantic gaps would automatically lead to more reuse. The fourth column is discussed in Section 6.3. It shows the number of proof obligations that must be handled to prove the correctness of each transformation.

| **Transformation** | **Reused within sequences** | **Reused for different target languages** | **Number of proof obligations** |
|---|---|---|---|
| Bidirectional to Unidirectional | no | yes | 1 |
| Clone Classes | yes | yes | 1 |
| Exclusive Channels | yes | no | 1 |
| Identify Channels | no | no | 1 |
| Lossless to Lossy | no | no | 74 |
| Merge Channels | no | no | 1 |
| Merge Objects | yes | no | 13 |
| Names to Arguments | no | no | 1 |
| Remove Classes | yes | yes | 1 |
| Strings to Integer | yes | no | 1 |
| Synchronous to Asynchronous | no | no | 4 and 34 |

Table 6.1: Endogenous model transformations

Because of their size and complexity, it is not possible to consider all transformations from Table 6.1. Instead, we select the two variants of the *Synchronous to Asynchronous* transformation to illustrate our approach. The difference in complexity of these two variants illustrates that more generic transformations employ more involved communication protocols for handling the introduced changes. One should search for the strongest possible constraints on the input models for such a transformation [6]. These constraints shall be realized in separate transformation steps that precede the more complex one in the transformation chain. This way, many unnecessary details are removed from the core part of the transformation. The simple variant of the *Synchronous to Asynchronous* transformation also described below, although simple, is still complex enough to illustrate all the details of our approach.

## 6.2.2 Refining Synchronous Communication

Synchronous communication is a typical example of a construct at a high level of abstraction that is often present in formal modeling languages. General-purpose programming languages, however, do not offer this concept. While SLCO allows for synchronous communication, the communication between controllers on the Lego Mindstorms platform is asynchronous. Therefore, synchronization should be realized with asynchronous interaction, introduced by correctly defined model transformations built around a properly defined communication protocol.

We defined two different transformations, $T^S_{as}$ and $T^G_{as}$, to replace one of the synchronous channels in a model by an asynchronous channel. To keep the observable behavior of the modeled system intact, this change requires and triggers further changes of the related classes, state machines, and transitions. Transformation $T^S_{as}$ applies to a restricted subset of models, but is simple and does not greatly increase the complexity of the produced model. In contrast, transformation $T^G_{as}$ can be applied to any SLCO model, but as a more complex protocol is introduced by the transformation, it adds more complexity to the produced model.

Both transformations require the following two constraints to hold for their input. First, the objects that communicate via the synchronous channel must be the only instances of their classes. Second, only a single pair of state machines from the two classes may communicate over the channel. We stress, however, that this does not limit their applicability. By means of the *Exclusive Channels* and *Clone Classes* transformations, any SLCO model can be transformed into a model with equivalent behavior that meets these constraints. Thus, instead of having more complicated transformations that first change models to meet these constraints and then replace synchronous communication by asynchronous communication, we opt for sequences of simpler transformations that have the same effect. The fact that the constraints hold can be used in the correctness proof of $T^S_{as}$ and $T^G_{as}$, which greatly simplifies these proofs.

In the remainder of this section, we provide a short description of the aforementioned transformations. An informal description is given in Section 3.5.1.1, and a more detailed description is given in Appendix C. For the rest of the section, we assume that in the model $m$, the synchronous channel $ch_s = chn()$ **sync from** $on_1.pn_1$ **to** $on_2.pn_2$ is to be replaced with an asynchronous one. Let object $o_i$ with name $on_i$ be an instance of class $cl_i$ with name $cn_i$ in model $m$, for $i = 1, 2$. We also assume, as explained above, that object $o_i$ is the only instance of class $cl_i$ and that state machine $sm_i$ with name $smn_i$ is the only state machine in $cl_i$ that uses channel $ch_s$, for $i = 1, 2$. Furthermore, we use $tr_s = tn_s$ **from** $ss_1$ **to** $ss_2$ **send** $sgn()$ **to** $pn_1$ to denote a transition of $sm_1$ of $cl_1$ that sends signals over $ch_s$ and $tr_r = tn_r$ **from** $sr_1$ **to** $sr_2$ **receive** $sgn()$ **from** $pn_2$ to denote a transition of $sm_2$ of $cl_2$ that receives signals over $ch_s$. Due to the uniqueness of the channel name and the aforementioned constraints, the transformation of the channel $ch_s$ induces a transformation of the classes $cl_1$ and $cl_2$ only. We show only the transformation of signals without arguments, but an extension to general signals is straightforward. In the remainder of this chapter, we deal with simplified SLCO models, as described in Section 3.7.

#### 6.2.2.1 Simple Transformation

Transformation $T^S_{as}$ modifies state machines by replacing some of their transitions. No essential changes are made to the other structures of a model. It is only applicable if, for every transition $tr_s$, there is no other transition with the same source state. For every transition $tr_s$ of $sm_1$ and for every transition $tr_r$ in $sm_2$, we define

$T^S_{as}(tr_s, pn_1) = \langle$
  $ss_{nw}$,
  $tn^1_s$ **from** $ss_1$ **to** $ss_{nw}$ **send** $ssgn()$ **to** $pn_1$
  $tn^2_s$ **from** $ss_{nw}$ **to** $ss_2$ **receive** $asgn()$ **from** $pn_1$
$\rangle$

$$T^S_{as}(tr_r, pn_2) = \langle$$
$sr_{nw}$,
$tn^1_r$ **from** $sr_1$ **to** $sr_{nw}$ **receive** $ssgn()$ **from** $pn_2$
$tn^2_r$ **from** $sr_{nw}$ **to** $sr_2$ **send** $asgn()$ **to** $pn_2$
$\rangle$,

where $ss_{nw}$ and $sr_{nw}$ are fresh state names, $tn^1_s$, $tn^2_s$, $tn^1_r$, and $tn^2_r$ are fresh transition names, $ssgn \equiv$ "$s_$" $+ sgn$, and $asgn \equiv$ "$a_$" $+ sgn$. In the transformed model, the new states are added to the appropriate state machines, and the transitions $tr_s$ and $tr_r$ are replaced by the newly generated transitions. For ease of reference, a graphical representation of the states and transitions in the definition above is given in Figure 6.2.

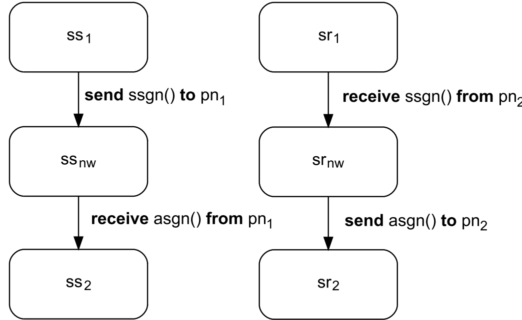



Figure 6.2: Partial state machines that illustrate $T^S_{as}$

#### 6.2.2.2 General Transformation

Transformation $T^G_{as}$ is more general than $T^S_{as}$, due to which it adds more complexity to the produced model. In this case, also classes are transformed, and new state machines are created. The restrictions we had on $T^S_{as}$ are removed, which means that $T^G_{as}$ can be applied to any $tr_s$ of $sm_1$ and any $tr_r$ of $sm_2$ as defined above. Transformation $T^G_{as}$ on transitions is defined as

$$T^G_{as}(tr_s, pn_1, vc_1) = \langle$$
$ss_3$ $ss_4$ $ss_5$ $ss_6$ $ss_7$,
$ts_1$ **from** $ss_1$ **to** $ss_3$ $vc_1 == 0$
$ts_2$ **from** $ss_3$ **to** $ss_4$ **send** $sgn(1)$ **to** $pn_1$
$ts_3$ **from** $ss_4$ **to** $ss_5$ $vc_1 == 2$
$ts_4$ **from** $ss_5$ **to** $ss_6$ **send** $sgn(3)$ **to** $pn_1$
$ts_5$ **from** $ss_6$ **to** $ss_2$ $vc_1 == 0$
$ts_6$ **from** $ss_7$ **to** $ss_1$ **send** $sgn(4)$ **to** $pn_1$
$ts_7$ **from** $ss_4$ **to** $ss_7$ $vc_1 := 2$
$\rangle$

$$
\begin{array}{l}
T^G_{as}(tr_r, pn_2, vc_2) = \langle \\
\quad sr_3\ sr_4\ sr_5\ sr_6\ sr_7, \\
\quad tr_1\ \textbf{from}\ sr_1\ \textbf{to}\ sr_3\ vc_2 == 1 \\
\quad tr_2\ \textbf{from}\ sr_3\ \textbf{to}\ sr_4\ \textbf{send}\ sgn(2)\ \textbf{to}\ pn_2 \\
\quad tr_3\ \textbf{from}\ sr_4\ \textbf{to}\ sr_5\ vc_2 == 3 \\
\quad tr_4\ \textbf{from}\ sr_5\ \textbf{to}\ sr_2\ \textbf{send}\ sgn(0)\ \textbf{to}\ pn_2 \\
\quad tr_5\ \textbf{from}\ sr_4\ \textbf{to}\ sr_1\ vc_2 == 4 \\
\quad tr_6\ \textbf{from}\ sr_7\ \textbf{to}\ sr_1\ \textbf{send}\ sgn(0)\ \textbf{to}\ pn_2 \\
\quad tr_7\ \textbf{from}\ sr_6\ \textbf{to}\ sr_7\ vc_2 := 3 \\
\quad tr_8\ \textbf{from}\ sr_1\ \textbf{to}\ sr_6\ vc_2 == 4 \\
\rangle,
\end{array}
$$

where $ss_j$ and $sr_j$, and $ts_j$ and $tr_k$ are fresh state and transition names, for $j = 3, \ldots, 7$ and $k = 3, \ldots, 8$. Variables $vc_1$ and $vc_2$ are discussed below. A graphical representation of the states and transitions in the definition above is depicted by the two partial state machines on the left of Figure 6.3.

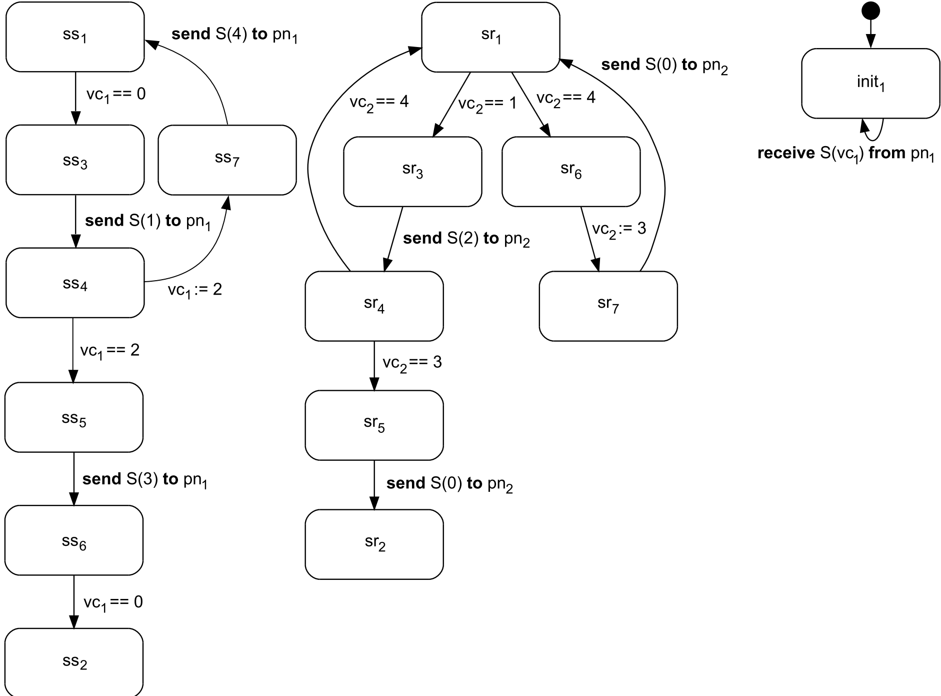



Figure 6.3: Partial state machines that illustrate $T^G_{as}$

In the transformed model, the new states are added to the appropriate state machines, and transitions $tr_s$ and $tr_r$ are replaced by the new transitions. Additionally, a fresh integer variable $vc_i$ and a state machine $reader_i$ are added to the classes $cl_i$, for $i = 1, 2$. Let $Tr^G_i$ be the sets of all $tr_s$ and $tr_r$-like transitions of $sm_i$, for $i = 1, 2$, $Sgn_1$ the set of all signal names occurring in the sending statements of transitions in $Tr^G_1$, and $Sgn_2$ the set of all signal names occurring in the reception statements of transitions in $Tr^G_2$. State

machine $reader_i$ is a result of applying function $Rsm$, defined as

$$Rsm(pn_i, vc_i, Sgn_i) = reader_i \textbf{ initial } init_i$$
$$[tsgn_i \textbf{ from } init_i \textbf{ to } init_i \textbf{ receive } sgn(vc_i) \textbf{ from } pn_i \mid sgn \in Sgn_i],$$

where $reader_i$ is a fresh state machine name, $init_i$ is a fresh state name, and $tsgn \equiv$ "$t_$" $+ sgn$. As defined, state machine $reader_i$ has a transition for every signal name $sgn$ from $Sgn_i$. On the right of Figure 6.3, an example of such a state machine is shown.

## 6.3 Correctness of Model Transformations

To reason about the correctness of a model transformation, we need a description of the behavior of models, a definition of the transformation, the possibility to check the correctness criteria, for instance by comparing the behavior of models, and the possibility to reason at the general language level rather than at the level of model instances. To cover all these aspects, a sufficiently expressive and flexible formalism has to be used. We decided to use labeled transition systems (LTSs) [93] as the underlying formalism to reason about the SLCO model transformations for several reasons. First, the LTS formalism is well-established and often used to describe the dynamic behavior of systems. Second, different equivalence relations between LTSs have been defined and used for the comparison of behavior. In Section 6.3.2, we use such an equivalence relation to relate the behavior of original and transformed models. Third, in our earlier work described in Chapter 5, an executable prototype of the semantics of SLCO is used to describe the behavior of models as LTSs. To utilize the experience gained by developing this prototype, we use the same formalism for the formal semantics of SLCO.

To generate an LTS representation of the behavior of a given model, we first formally define the dynamic semantics of SLCO in the form of structural operational semantics (SOS) [93]. Then, reasoning at the level of LTS representations of the behavior of models, we define our criterion for correctness of model transformations. In the following section, an overview of the formal semantics of SLCO required for the correctness proofs is given. A more detailed description is presented in Appendix B.

In contrast to the aforementioned prototype, the formal semantics does not take time and successful termination into account. We decided not to use the prototype itself as a definition of the semantics of SLCO, to abstract from the details of its implementation. Although the executable prototype and the formal semantics assign equivalent LTSs to a given SLCO model without delay statements and final states, the rules that define the formal semantics of SLCO do not correspond directly to the rules used to implement the prototype. This is caused by the fact that the formal semantics define the possible behavior of a transition in any context, whereas the executable prototype defines the behavior of transitions in relation to other transitions.

In the sequel, we use the following conventions and notation. If $lt$ is a sequence, we abuse the notation and write $lt$ for the set of elements of $lt$. Thus, we write $elt \in lt$ for "there is an element $elt$ in sequence $lt$". A number of functions are defined and used. The update of function $f$ such that $f(x) = a$ is denoted by $f[a/x]$.

### 6.3.1 Operational Semantics of SLCO

For an arbitrary model $m$, the SOS rules that define the semantics of SLCO generate the complete behavior of the model in the form of an LTS. An LTS is a tuple $(S, \Lambda, \rightarrow, i)$, where $S$ is a set of configuration, $\Lambda$ is a set of labels, $\rightarrow \subseteq S \times \Lambda \times S$ is a ternary relation of labeled transitions, and $i \in S$ is the initial configuration. In our case, for an SLCO model $m = mn\ obj^*\ class^*\ chan^*$, with $VN$ the set of all variable names occurring in $m$, $SMN$ the set of all state machine names of $m$, $SN$ the set of all state names of these state machines, and $CE$ the set of all constant expressions, the configurations of the LTS generated for $m$ are tuples $\langle m, s_{sms}, v_{glob}, v_{loc}, b\rangle$, where $s_{sms} : ON \nrightarrow (SMN \nrightarrow SN)$ is a function that indicates current states of the state machines of the objects in $m$, $v_{glob} : ON \nrightarrow (VN \nrightarrow CE)$ is an evaluation function that assigns values to the (global) variables of the objects in $m$, $v_{loc} : ON \nrightarrow (SMN \nrightarrow (VN \nrightarrow CE))$ is an evaluation function that assigns concrete values to the (local) variables of the state machines of the objects of model $m$, and $b : (CHN \times ON \times ON) \nrightarrow (SGN \times SEQ(CE)) \cup \{\mathbf{nil}\}$ represents the content of the one-place buffers corresponding to the asynchronous channels in $m$. The content of a buffer can be **nil**, denoting an empty buffer, or a tuple consisting of a signal name and a sequence of constant expressions. Two buffers are associated to each bidirectional, asynchronous channel, one for each direction.

For the LTS of $m$, $LTS(m)$ in short, the initial configuration conforms to the following constraints. First, all the buffers assigned to asynchronous channels are initialized to **nil**. Second, all the state machines are in their initial state. Third, all variables are initialized to values respecting their types.

The transitions of $LTS(m)$ are obtained as the least relation deduced from the SOS rules. A transition can be labeled $\epsilon$, $vn := ce$, $sgn(ce^*)$, **send** $sgn(ce^*)$, **receive** $sgn(ce^*)$, or **lost** $sgn(ce^*)$, where $vn$ denotes a variable name, $ce$ a constant expression, $sgn \in SGN$ a signal name, and $ce^*$ a sequence of constant expressions. Transitions labeled $\epsilon$ correspond to transitions in SLCO models without a statement or transitions that have a blocking statement represented by a Boolean expression. Transitions labeled $vn := ce$ represent assignments. The label $sgn(ce^*)$ represents synchronous communication between two objects, whereas the labels **send** $sgn(ce^*)$, **receive** $sgn(ce^*)$, and **lost** $sgn(ce^*)$ denote sending, receiving, and losing signals during asynchronous communication.

The overall behavior of a model is defined by the behavior of its composite elements, the objects and the channels. The statements that the objects can execute and the way they interact via the channels determine the dynamics of the model. The contributed activities of an object are deduced from the specification of its class, and the activities of a class are deduced from the specifications of its state machines. At the most elementary level of the structure, the behavior of each state machine is derived from its transitions.

$$\frac{\langle e^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} ce^*}{\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ \mathbf{send}\ sign(e^*)\ \mathbf{to}\ pn, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\mathbf{send}\ sgn(ce^*)\ \mathbf{to}\ pn}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle}$$

$$\frac{\langle ce^*, vn^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{SUB} \langle v''_{vars}, v'''_{vars}\rangle, \quad \langle e, v''_{vars}, v'''_{vars}\rangle \Rightarrow_{EXP} \mathbf{true}}{\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ \mathbf{receive}\ sgn(vn^* \mid e)\ \mathbf{from}\ pn, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\mathbf{receive}\ sgn(ce^*)\ \mathbf{from}\ pn}_{TRANS} \langle sn', v''_{vars}, v'''_{vars}\rangle}$$

Figure 6.4: A subset of all deduction rules for transitions

Figure 6.4 shows some of the SOS rules that relate a single transition in an SLCO model to a transition on the LTS level. If an expression $e$ evaluates to constant expression $ce$ with respect to evaluation functions $v_1$ and $v_2$, we write $\langle e, v_1, v_2 \rangle \Rightarrow_{EXP} ce$. Similarly, we write $\langle e^*, v_1, v_2 \rangle \Rightarrow_{EXPS} ce^*$ for a sequence of expressions $e^*$. Finally, we use $\langle ce^*, vn^*, v_1, v_2 \rangle \Rightarrow_{SUB} \langle v_1', v_2' \rangle$ to denote the update of evaluation functions $v_1$ and $v_2$ to $v_1'$ and $v_2'$, such that variables $vn^*$ are mapped to the values $ce^*$. The first rule in Figure 6.4 deals with statements that send signals. The evaluation functions $v_{vars}$ and $v'_{vars}$ map values to variables and are used to evaluate the expressions that form the arguments of the signal. The second rule in Figure 6.4 deals with conditional signal reception. The rule specifies that the expression that forms the condition is evaluated using the possibly updated evaluation functions. If the condition evaluates to **true**, the transition is enabled.

$$\frac{\begin{array}{c} trans \in trans^*, \\ \langle trans, s_{sms}(smn), v_{vars}, v_{sms}(smn) \rangle \xrightarrow{l}_{TRANS} \langle sn, v'_{vars}, v''_{vars} \rangle, \\ s'_{sms} = s_{sms}[sn/smn], \quad v'_{sms} = v_{sms}[v''_{vars}/smn] \end{array}}{\langle smn\ \ var^*\ \ states\ \ trans^*, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}$$

Figure 6.5: Deduction rule for state machines

The behavior of state machines is defined in terms of the behavior of their transitions, as shown in Figure 6.5. The rule defines that if one of the transitions of state machine $sm$ can perform an action represented by $l$, then the state machine can perform the same action.

$$\frac{sm \in sm^*, \quad \langle sm, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}{\langle cn\ \ var^*\ \ port^*\ \ sm^*, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle}$$

Figure 6.6: Deduction rule for classes

The rule in Figure 6.6 shows how the enabled transitions, derived from the SOS rules for state machines, are lifted up to the level of a class. It also shows implicitly that all state machines of a considered class are inspected. The symbolic function $s_{sms}$ keeps track of the current states of the state machines, while $v_{sms}$ maps (local) variables to their value for the state machines of each object, and $v_{vars}$ maps the (global) variables at the level of the objects to their values.

Objects behave as specified by their class. In a composition, objects participate and interact as described by the SOS rules for compositions of objects, some of which are given in Figure 6.7. Every non-synchronizing transition of one of the objects enabled for execution in the current configuration is executed by the composition of the objects, and the functions are updated accordingly. A non-synchronizing transition that receives signals over an asynchronous lossless channel is captured by the first rule in Figure 6.7. The second rule describes synchronization of two objects via a synchronous channel.

Finally, the rule in Figure 6.8 defines the behavior of a model in terms of its objects, classes, and channels. Each transition from one configuration to another is derived from the rules discussed above.

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossless from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1], \quad b(\langle chn, on_1, on_2\rangle) = \textbf{nil}, \quad b' = b[\langle sgn, ce^*\rangle/\langle chn, on_1, on_2\rangle] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{\textbf{send}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v_{glob}, v_{loc}, b'\rangle}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \quad v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle}$$

Figure 6.7: A subset of all deduction rules for compositions of objects

$$\frac{\begin{array}{c} m \equiv mn\ obj^*\ class^*\ chan^*, \\ \langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle \end{array}}{\langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{MODEL} \langle m, s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle}$$

Figure 6.8: Deduction rule for models

### 6.3.2 Correctness of the $T_{as}^S$ transformation

The operational semantics of SLCO generates an LTS representation of the model dynamics, for a given model and its initialization. Thus, to reason about the correctness of and property preservation by a model transformation, we need to compare the behaviors of two models, one before and one after the transformation, represented as LTSs.

A wide range of equivalence relations on LTSs have been proposed [45]. Some of them, such as strong bisimilarity, are appropriate for concrete behavior, when every action of the system is observable. For some of the defined SLCO transformations, this is indeed sufficient. However, these relations are often too fine when part of the behavior is preferred to be abstracted away and considered unobservable. Some of the SLCO model transformations, as explained in the previous sections, add more detail to the behavior and therefore, some parts of the behavior introduced by the transformation need to be abstracted away to mimic the behavior before the transformation. In view thereof, we choose branching bisimilarity [47] as the equivalence relation we use for the correctness criterion. Branching bisimilarity is a relation between configurations of LTSs for which some transitions are considered internal (or unobservable), which is represented by labeling them with $\tau$ ($\tau \notin \Lambda$). Intuitively, two configurations are branching bisimilar if every transition step that can be executed in one configuration can be mimicked in the other, possibly after a finite number of internal steps. Branching bisimilarity can be formally defined as follows.

**Definition 1.** *For two LTSs $L_1 = (S_1, \Lambda_1, \rightarrow_1, i_1)$, $L_2 = (S_2, \Lambda_2, \rightarrow_2, i_2)$ a relation $R \subseteq S_1 \times S_2$ is called a branching bisimulation relation if the following conditions are met, for all $s \in S_1$ and $t \in S_2$ such that $R(s,t)$.*

1. *If $s \xrightarrow{a} s'$ in $L_1$, then either*
   - *$a = \tau$ and $R(s', t)$, or*
   - *for some $n \geq 0$, there exist $t_1, \ldots, t_n$ and $t'$ in $S_2$ such that $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} t_n \xrightarrow{a} t'$ in $L_2$, $R(s, t_n)$ and $R(s', t')$;*
2. *If $t \xrightarrow{a} t'$ in $L_2$, then either*
   - *$a = \tau$ and $R(s, t')$, or*
   - *for some $n \geq 0$, there exist $s_1, \ldots, s_n$ and $s'$ in $S_2$ such that $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n \xrightarrow{a} s'$ in $L_1$, $R(s_n, t)$ and $R(s', t')$.*

*LTSs $L_1$ and $L_2$ are branching bisimilar if there exists a branching bisimulation relation $R$ for $L_1$ and $L_2$ such that $R(i_1, i_2)$.*

Branching bisimilarity possesses many useful properties, one of which is that related models possess the same properties that can be expressed in the temporal logic CTL$^*$ without the next state modality [29]. In our case, this means that if a certain property has been established for the source model, which is usually much smaller than its implementations and thus easier to analyze, and if we apply a (well-composed) sequence of model transformations for which our correctness criterion hold, then the generated model inherits the property by construction. The same holds for all intermediate models. Thus, our correctness criterion for model transformation provides effective and efficient generation of implementations that are correct-by-construction.

**Definition 2.** *Let $T$ be an SLCO model transformation such that for any SLCO model $m$ to which $T$ applies and for a given initialization of $m$, there is a renaming $\rho$ of the labels of the $LTS(T(m))$ such that $LTS(m)$ and $\rho(LTS(T(m)))$ are branching bisimilar. Then, $T$ is a correct model transformation.*

The renaming function $\rho$ in the definition above is needed to rename some labels into $\tau$, but also to unify, if needed, the labels of transitions that are supposed to mimic each other. For example, when replacing a synchronous channel with an asynchronous one, a small protocol is employed. On the LTS level, one of the labels that represent the steps performed as part of this protocol corresponds to the label that represents the synchronization in the original situation. The renaming function is used to ensure that these labels are the same and to hide the labels that represent the other steps.

In the remainder of this section, we discuss the main lines of the correctness proof for the simple variant of the *Synchronous to Asynchronous* transformation, $T^S_{as}$. We chose this transformation because its proof has all the important aspects that need to be taken into account, yet the established relation between configurations is simple enough to be given completely. Besides transformation $T^S_{as}$, there are four more transformations that require a substantial amount of cases to be considered for their correctness proof. The fourth column of Table 6.1 in Section 6.2 lists the number of proof obligations for each transformation. The last row lists two numbers, one for each version of the *Synchronous to Asynchronous* transformation. Fortunately, the correctness proofs of the majority of transformations involve a single proof obligation only, relating each configuration for

input models to exactly one equivalent configuration for output models. This is a clear benefit of designing sequences of transformations that are as fine-grained as possible.

Referring back to the definition of $T^S_{as}$ in Section 6.2.2, let $Sgn$ be the set of all signal names used in the sending and receiving statements of all $tr_s$ and $tr_r$-like transitions. We define a label-renaming function $\rho$ as follows, for every $ssgn \equiv$ "$s_$" + $sgn$ and $asgn \equiv$ "$a_$" + $sgn$ such that $sgn \in Sgn$.

- $\rho(\textbf{send}\ ssgn()) = \tau$
- $\rho(\textbf{receive}\ ssgn()) = sgn()$
- $\rho(\textbf{send}\ asgn()) = \tau$
- $\rho(\textbf{receive}\ asgn()) = \tau$

Renaming $\rho$ is straightforwardly extended on $LTS(T^S_{as}(m, chn))$. By renaming the label **receive** $ssgn()$ to $sgn()$, we indicate that these two labels represent successful communication. The other labels are renamed to $\tau$ because they represent the implicit synchronization in the source model and should result in unobservable behavior of the target model.

**Theorem 1.** *For a given SLCO model* $m = mn\ class^*\ obj^*\ chan^*$ *and a channel* $ch_s = chn()$ ***sync from*** $on_1.pn_1$ ***to*** $on_2.pn_2 \in chan^*$*,* $T^S_{as}(m, chn)$ *is a correct model transformation.*

*Proof.* We need to show that $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar. As usual, the main difficulty of the proof lies in properly relating the configurations from $LTS(m)$ and those from $\rho(LTS(T^S_{as}(m, chn)))$. Before we define the relation, it is worth noticing that each configuration of $LTS(m)$ is also a configuration of $\rho(LTS(T^S_{as}(m, chn)))$, but for the latter the buffer function is extended over the triples $\langle chn, on_1, on_2\rangle$ and $\langle chn, on_2, on_1\rangle$.

Let configuration $cf = \langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle$ be a configuration of $LTS(m)$ and let $cf' = \langle T^S_{as}(m, chn), s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle$ be a configuration of $\rho(LTS(T^S_{as}(m, chn)))$. We define a relation $R$ between the configurations as follows: $(cf, cf') \in R$ if and only if $v_{glob} = v'_{glob}$, $v_{loc} = v'_{loc}$ and

1. $s_{objs} = s'_{objs}$, $b'(\langle chn, on_1, on_2\rangle) = \textbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = \textbf{nil}$, and $b' = b$ otherwise, or

2. $s_{objs}(on_1)(smn_1) = ss_1$, $s'_{objs}(on_1)(smn_1) = ss_{nw}$, $s_{objs} = s'_{objs}$ otherwise, $b'(\langle chn, on_1, on_2\rangle) = (ssgn, \varepsilon)$, $b'(\langle chn, on_2, on_1\rangle) = \textbf{nil}$, and $b' = b$ otherwise, only if there is a $tr_s$-like transition from $ss_1$ with signal name $sgn()$, or

3. $s_{objs}(on_1)(smn_1) = ss_2$, $s_{objs}(on_2)(smn_2) = sr_2$, $s'_{objs}(on_1)(smn_1) = ss_{nw}$, $s'_{objs}(on_2)(smn_2) = sr_{nw}$, $s_{objs} = s'_{objs}$ otherwise, $b'(\langle chn, on_1, on_2\rangle) = \textbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = \textbf{nil}$, and $b' = b$ otherwise, only if there is a $tr_s$-like transition in $sm_1$ to $ss_2$, and there is a $tr_r$-like transition in $sm_2$ to $sr_2$, or

4. $s_{objs}(on_1)(smn_1) = ss_2$, $s'_{objs}(on_1)(smn_1) = ss_{nw}$, $s_{objs} = s'_{objs}$ otherwise, $b'(\langle chn, on_1, on_2\rangle) = \textbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = (asgn, \varepsilon)$, and $b' = b$ otherwise, if there is a $tr_s$-like transition to $ss_2$ with signal $sgn()$.

Next, for each pair of configurations, we have to show that they can mimic each other, using the SOS rules. For example, let us consider case **2.** If $cf \xrightarrow{l}_{MODEL} cf_1$, for some label $l$ and a configuration $cf_1$ of $LTS(m)$, then either $l \not\equiv sgn()$ or $l \equiv sgn()$. In the first

case, this transition certainly does not involve state machine $sm_1$ of object $o_1$, since it can only synchronize in this configuration. In this case, the same state machine(s) of the same object(s) can induce the same transition $cf' \xrightarrow{l}_{MODEL} cf_1'$, and since the updates of the functions do not change $s_{objs}(o_1)(sm_1)$ and $s'_{objs}(o_1)(sm_1)$, it follows that $(cf_1, cf_1') \in R$.

If $l \equiv sgn()$, then $sm_2$ has to be in a state $sr_1$ for some $tr_r$-like transition, according to the SOS rule concerning synchronous communication in Figure 6.7. According to the SOS rule for an asynchronous signal reception in the same figure, $cf' \xrightarrow{sgn()}_{MODEL} cf_1'$, where the label $sgn()$ in $\rho(LTS(T^S_{as}(m, chn)))$ is the result of renaming the label **receive** $ssgn()$ of $LTS(T^S_{as}(m, chn))$. Furthermore, in $cf_1$, $sm_1$ is in state $ss_2$ and $sm_2$ is in state $sr_2$. In $cf_1'$, $sm_1$ is in state $ss_{nw}$ and $sm_2$ is in state $sr_{nw}$. According to **3.**, $(cf_1, cf_1') \in R$. Lifting the constraint on $sm_1$ and allowing it to have other transitions besides the one that sends a synchronous signal in state $ss_1$ breaks bisimilarity of $R$.

For each transition $cf' \xrightarrow{l}_{MODEL} cf_1'$ that is enabled in configuration $cf'$ we can show in a similar way that it is either also enabled in configuration $cf$ or that it is mimicked by a transition $cf \xrightarrow{sgn()}_{MODEL} cf_1$. By a careful inspection of transitions generated by the SOS rules for the other three cases of pairs of $R$-related configurations, we can prove that $R$ is indeed a branching bisimulation that relates $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$. □

A more detailed description of this proof is given in Appendix D.

## 6.4 Related work

Various aspects of the correctness of model transformations have been considered, and different approaches have been proposed. Giese et al. relate input and output models during the specification of a transformation and then use a theorem prover to show semantic equivalence between the input and output of this transformation [44]. The source and target language discussed are relatively small, leading to a more straightforward transformation compared to the sequences of transformations we consider. However, the use of a theorem prover to automate parts of the correctness proofs has clear advantages over manual proofs. The approach of Schätz also uses a theorem prover for assistance with correctness proofs [101]. In this case, properties that are proved for the given model transformation are more of a structural nature. It is interesting that here the author advocates the advantages of having a single homogenous formalism for description of transformations, which we also see advantageous in our approach. The framework proposed by Varró allows for defining a set of graph transformation rules to describe the operational semantics of a DSML, which is used, similar to our approach, to generate an LTS representation of models in the DSML, which then can be model checked [109]. However, the translation framework works only on a particular given model instance of the language, while we aim at general results at the level of the entire language.

Instance-based verification of model transformations is described also by Karsai and Narayanan [65]. Their approach entails generating a certificate for each model that is transformed. These certificates are used to show that the model transformation preserves certain properties for the given input model, but cannot be used to show that properties are preserved for arbitrary input models.

The approaches described by Ehrig and Ermel [36], and Hülsbusch et al. [57] are most closely related to the work presented in this chapter. Ehrig and Ermel consider preservation of behavior by model transformations [36]. Besides the models in the source language, also the language semantics is transformed, and the result is compared with the semantics of the target language. The paper states conditions that input models

and model transformations should fulfill to preserve the semantics. Hülsbusch et al. [57] consider the correctness of model transformations stated in terms of a bisimulation relation. Here, the languages are first given operational semantics in terms of graph-transformation rules. Although our approach to the correctness of transformations is similar to this one, the two languages considered in their work are simpler than the language we used to demonstrate our approach.

## 6.5 Conclusions and Future Work

To address research question $RQ_4$, we defined a formal framework for reasoning about the correctness of the endogenous model transformations related to SLCO, which is illustrated in this chapter. Using this framework, we can assess whether sequences of transformations are well-composed, and whether individual transformations are provably correct. By designing sequences of transformations that are as fine-grained as possible, we improved the reusability of these transformations, both between sequences of transformations and within such sequences. Furthermore, this design decision increased the number of transformations with straightforward correctness proofs and reduced the number of proof obligations for the proofs of the larger transformation steps.

Given the large number of straightforward proofs, we consider investigating the application of automated theorem proving to be a promising direction for future research. Furthermore, the work presented in this chapter does not take successful termination and time into account. These problems also provide opportunities for future research. Finally, we want to investigate the generalization of the approach to other DSMLs.

Chapter 7

# Evolution of a Domain-Specific Modeling Language

*In this chapter, we describe the evolution of the Simple Language of Communicating Objects (SLCO) and the corresponding model transformations. These transformations were developed one at the time, simultaneously with the language itself, while preserving the validity of the previously developed ones. The simultaneous development of the language and the transformations has led to an iterative evolution of SLCO. Furthermore, the description of the semantics of SLCO changed from a transformational definition to a definition based on an executable state-space generator. Afterwards, a formal definition of its semantics was given. Switching from one form of defining the semantics to another also led to adaptations of the language. In this chapter, we describe our development process and the causes for the evolution of the language and its transformations.*

## 7.1 Introduction

The Simple Language of Communicating Objects (SLCO) gradually evolved from a slightly modified subset of the UML suited for performance analysis to a domain-specific modeling language (DSML) that offers simulation, verification, and execution of models. In this chapter, we describe our experiences with the process of developing this DSML and the corresponding model transformations. By studying the development of a small and easily changeable language and documenting its evolution, we started to investigate the feasibility of a development process for DSMLs in which changes to one artifact have less influence on other artifacts.

The literature provides guidelines for developing domain-specific languages (DSLs) [31, 32, 80] and tools that support DSL evolution [43, 105]. There have been numerous reports on evolution of DSLs. For example, the changes to the SDF language, a DSL used for syntax definition, are described by Visser [110]. Van Beek et al. describe the evolution of Chi, a language for modeling and simulating hybrid systems [15]. The evolution of a language used for interchanging models of hybrid systems is described by Van Beek et al. [14]. In most of these publications, however, only the changes themselves are described, and the reasons for these changes are only hinted upon. On the underlying reasons

for DSL evolution literature is scarce and conclusions are scattered. Additionally, most analysis of DSL evolution provide an a posteriori report only. In this chapter, we discuss our experiences with an evolving DSML during its iterative design and address research question $RQ_5$.

> **$RQ_5$:** *What are the main influences on the design of a DSML and the corresponding model transformations?*

As described in Chapter 3, we designed a DSML for modeling systems consisting of concurrent, communicating objects. The structure of a system is modeled using classes, and their behavior is modeled by state machines. Simultaneously to the development of the DSML, we implemented a number of model transformation to different formalisms: one for simulation, one for execution, and one for verification. These model transformations were developed consecutively. Each time a transformation to a new target platform was added, the functionality offered by the existing transformations remained intact.

Furthermore, the way in which the semantics of SLCO were defined changed over time, which also had its influence on the rest of the definition of the language and its transformations. At first, the semantics of SLCO were defined by means of a transformation to the Parallel Object-Oriented Specification Language (POOSL) [108]. This transformational description was then replaced with another transformational description based on labeled transition systems. Later, a formal definition of the operational semantics of SLCO was given, to enable formal reasoning about the correctness of the model transformations.

We describe the development process and indicate how our DSML has evolved during this process. We focus on language evolution only and not on the co-evolution of models specified in the DSML such as described in [24]. Even though our DSML is small in terms of the provided number of modeling constructs, we expect that the lessons learned are applicable to projects involving larger DSMLs as well. In our discussion of related work in Section 7.4, we show that the conclusions drawn by others overlap with our own.

The remainder of this chapter is structured as follows. In Section 7.2, we describe the development process of our DSML and the accompanying model transformations. Section 7.3 describes the evolution the language has undergone. Related work is discussed in Section 7.4. In Section 7.5, we draw conclusions and give directions for further research.

## 7.2 Development Process

The language and the sequences of model transformations described in Chapter 3 originate from research aimed at performance analysis of UML models. The goal of this research was to be able to simulate and analyze UML models, developed using an intuitive, graphical syntax, without the need for modelers to learn the syntax and semantics of a formal language for simulation. Because only a small subset of the UML was used, and some additions and other changes were needed, we decided to create a new DSML once the original research project was finished. This way, we no longer had to deal with UML's very large metamodel. The new DSML was named the Simple Language of Communicating Objects (SLCO). Once it was defined, a model transformation was implemented to enable simulation of its models using POOSL [108], which is described in Section 3.5.2.1. By means of this transformation, SLCO models could be simulated, and the performance of the systems they describe could be analyzed. Second, a model transformation was implemented to enable execution of the models on the Lego Mindstorms platform[1], which

[1] http://mindstorms.lego.com/

is described in Section 3.5.2.2. Executing the code generated from certain models revealed bugs in these models that originated from unforeseen interleavings of concurrent objects. These bugs were not encountered during simulation. To detect this kind of problems, a third model transformation was implemented to enable verification of the models using Spin [55], which is described in Section 3.5.2.3.

At first, the transformation from SLCO to POOSL, a language whose semantics is formally defined, served as a transformational definition of SLCO's semantics. However, to reason about the correctness of the model transformations related to SLCO, a formal and more concise definition of its semantics was needed. Thus, as a starting point for a formal operational semantics for SLCO, an executable prototype of the semantics was implemented, which is described in Chapter 5. This prototype made it possible to experiment with various alternative design decisions. Once the prototype reached a stable state, we defined the formal semantics of SLCO based on the prototype, which is described in Appendix B.

Adding new target platforms to an existing DSML and changing the way its semantics are defined led to changes in the DSML itself. In the remainder of this chapter, we describe these changes and indicate their causes.

## 7.3 Evolution

Van Deursen et al. identify three phases in the development of a DSL: the analysis phase, the implementation phase, and the phase in which the DSL is used [32]. Mernik et al. split the analysis phase into an analysis and a design phase [80]. Because the development of our language is an iterative process, the phases are revisited each time the language is extended. The evolution of the language and the transformations has been influenced by a number of roles, each of which is responsible for performing certain activities that belong to the four phases in the development of a DSL mentioned above. We describe the evolution of our DSML and the accompanying transformations in terms of the activities performed during these four phases. The remainder of this section starts with a description of the roles and the activities that belong to these roles. After these descriptions, the major changes made in both the language and the transformations are listed. At the end of this section, we cluster these changes and distinguish four types of causes for evolution.

### 7.3.1 Roles and Activities

The design of our DSML has been influenced by a number of roles. Table 7.1 shows in which phases of the development each of the roles participate. Although every role has its own separate tasks, these tasks greatly depend on each other, which leads to interaction between the roles. In the evolution of our DSML, language designers, platform experts, modelers, and software developers have played a role.

| **Development phases** | **Roles** |
|---|---|
| Analysis | Language designer<br>Platform expert |
| Design | Language designer |
| Implementation | Software developer |
| Use | Modeler |

Table 7.1: Development phases and the corresponding roles

Table 7.2 shows how the aforementioned roles are related to activities and the number of persons that fulfilled the roles. The actual number of modelers is larger than shown in the table. SLCO has been used in a number of student assignments in which students had to create models. However, little to no feedback was acquired from the students. Because their influence on the evolution of SLCO is negligible, they are not explicitly mentioned in the table. Below, we describe how the roles and activities relate.

In our development process, language designers are involved in two activities. First, they perform a domain analysis to investigate which aspects of the problem domain need to be incorporated in the language. After the domain analysis is completed and all the concepts that need to be incorporated in the DSML have been identified, the language designers are responsible for defining the syntax and semantics of the language.

| **Role** | **Persons** | **Activities** |
|---|---|---|
| Language designer | 4 | Domain analysis<br>Defining the language |
| Platform expert | 4 | Domain analysis<br>Defining the mapping from SLCO to a platform<br>Interpreting models |
| Modeler | 3 | Creating models<br>Applying transformations<br>Interpreting models |
| Software developer | 2 | Implementing the mappings<br>Implementing editors for SLCO models |

Table 7.2: Roles and the corresponding activities

SLCO models are transformed to a number of platforms. Support for these target platforms was added one at a time, and in some cases, adding a new platform changed the application domain of the language. This is caused by our decision to extend the language whenever necessary such that it is suited for modeling on both high and low levels of abstraction. For example, to be able to use SLCO to model systems on the level of abstraction offered by NQC, the concept of asynchronous communication was added to the language. Each time changes like this have to be made, the platform expert performs an additional analysis of the domain of the language. Platform experts are also responsible for the mapping from SLCO to new target platforms. This mapping defines how constructs of the DSML are mapped onto constructs available for the target platform. In some cases, not all constructs need to have an equivalent counterpart in the target platform. The language and tools offered by the target platform have a certain purpose, and some constructs of SLCO may be irrelevant for the purpose of the target platform. Lastly, platform experts may be involved in the interpretation of models. For example, a Spin expert was asked to analyze a number of SLCO models that suffered from state-space explosion after translating them to Promela and processing them with Spin. Based on the advice of this expert, both the language and the model transformations were changed. In our case, a POOSL expert, Spin experts, and NQC experts participated in the development of the language.

The modeler is the end-user of a DSML. Besides creating models and generating new models by applying transformations, the modeler analyzes and interprets models. To simulate an SLCO model, for example, the modeler can transform this model to an equivalent POOSL model. This POOSL model can then be simulated, after which the results of the simulation have to be interpreted to determine their relevance for the

original SLCO model.

Given a mapping from SLCO to a target platform designed by a platform expert, software developers are responsible for the translation of the conceptual mapping to an actual implementation of the transformation. Furthermore, the software developers implement the editors required for creating and editing models.

### 7.3.2 Evolution of the Language

While performing the activities described in Section 7.3.1, various features were identified and added to the language. Table 7.3 shows how the activities are related to these features. Each of the features is discussed in the remainder of this section.

| Activity | Language features |
|---|---|
| Domain analysis | Concurrent, communicating objects<br>Simple imperative constructs<br>Delays<br>Asynchronous signals<br>Lossy channels |
| Defining the language | UML-like syntax<br>Synchronous signals<br>Subset for formal semantics |
| Interpreting models | Conditional signal reception<br>Combination of trigger and guard<br>Local variables<br>Structure diagrams<br>Additional operators and types<br>Incoming transitions for initial states<br>Outgoing transitions for final states<br>Fine-grained sequences of transformations |
| Creating models | Initial values of variables<br>Identifying signals by name<br>Textual syntax<br>Generalization of statements, triggers, and guards |
| Implementing the mappings | Explicit channel types<br>Restricted form of conditional signal reception |

Table 7.3: Desired language features identified during each of the activities

From an abstract point of view, the problem domain for which SLCO was designed consists of concurrent, communicating objects. The most important requirement for our DSML is therefore that it can describe such objects at an appropriate level of abstraction. The initial application area of SLCO was performance analysis of interoperating software components implemented with an imperative programming language. Besides communicating with each other, these components perform simple calculations. To be able to express the calculations performed by the components, a small number of basic imperative constructs were added to the language. Furthermore, to enable performance analysis, time needed to be incorporated into the language, which was achieved by adding delay statements. All of these aspects of the language were identified during the initial analysis of the problem domain. At a later stage, the aspiration to execute SLCO models on the Lego Mindstorms platform gave rise to a second analysis of the problem domain. Communication between RCXs, the controllers of the Lego Mindstorms platform, occurs

using infrared signals. These signals are asynchronous, and the channel used for this type of communication is unreliable. Initially, communication in SLCO could only occur using synchronous channels over reliable channels, as explained below. To enable a straightforward mapping from SLCO to NQC, the characteristics of both languages should be aligned. Therefore, SLCO was extended with asynchronous signals and lossy channels as a result of the additional domain analysis.

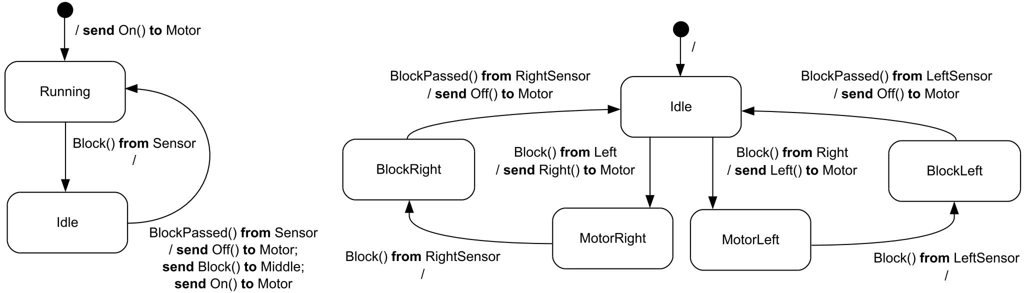



Figure 7.1: Syntax that distinguishes triggers, guards, and statements

One purpose of SLCO is using the language for documentation. For this reason, a graphical syntax resembling the well-known syntax of the UML was chosen while defining the language. Another design decision we made for this reason is offering communication via synchronous signals. This form of communication leads to concise models, which increases the understandability of models and thus increases their value as documentation. In Figure 7.1, two state machines are shown using the graphical syntax that was inspired by the UML. Over time, this syntax has changed a little, as is explained below. Finally, to simplify the definition of SLCO's formal semantics, a sublanguage has been identified, which is described in Section 3.7. Because each of the constructs that are missing from this sublanguage can be expressed in terms of its other constructs, SLCO's semantics can be defined completely by defining the semantics of this sublanguage only.

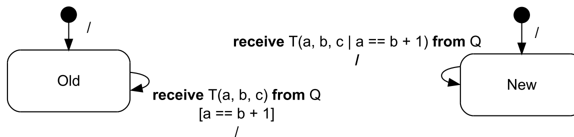



Figure 7.2: Evolution of conditional signal reception

Many of the changes to the language were inspired by problems encountered during the interpretation of models. During the interpretation of Promela models, for example, we noticed that the state space generated as part of the model-checking process could be reduced by adapting the language. Because reducing the state space improves verification performance, we adapted SLCO by introducing conditional signal reception as a language construct. Conditional signal reception can be used to specify that a signal can only be received if its arguments adhere to a certain condition. The leftmost transition from state *Old* to itself in Figure 7.2, for example, will only be taken when a signal $T$(a, b, c) is received for which the condition $a == b + 1$ holds. The original syntax used for this construct was again inspired by the UML. The fact that the variables used in the guard $a == b + 1$ refer to the values received as arguments of signal $T$ proved to be a

source of confusion that was addressed at a later stage. The condition was incorporated into the language construct for signal reception, as shown on the right of Figure 7.2.

During the interpretation of SLCO models, we noticed that some variables were used only locally in the state machines, but were defined globally as variables of the class containing the state machines. To improve the readability of the models, we introduced the concept of local variables. However, this increase in readability is accompanied by a decrease in modifiability when using the standard tree view editor provided by the Eclipse Modeling Framework (EMF) [106] for editing models. When referring to a variable in this tree view editor, the container of a variable is not shown in the list of variables that can be referred to. This makes it hard to distinguish between two variables with the same name but with a different container. The readability of models was also improved by extending the graphical syntax with structure diagrams. These diagrams make it possible to visualize all aspects of an SLCO model. They provide information about models that was missing from the communication and behavior diagrams, such as the scope and initial value of variables. Another improvement in terms of readability and modifiability we made after interpreting SLCO models is adding new operators and types to the language.

Once the state spaces of SLCO models could be visualized with the executable prototype of SLCO's semantics presented in Chapter 5, the fact that initial states were not allowed to have incoming transitions and final states were not allowed to have outgoing transition showed to clutter the graphs that represented the state spaces. Therefore, we removed these restrictions and changed the way initial and final states are shown in behavior diagrams. Previously, both the solid black circles as well as the rounded rectangles in behavior diagrams, such as the ones shown in Figure 7.2, represented states. After the aforementioned restrictions were removed, a solid black circle no longer represents a separate state. Instead, the black circle is now only used to indicate which of the rounded rectangles represents an initial state. The way final states are displayed changed similarly. Table 7.3 also indicates that the transformations have changed as a result of problems that occurred while interpreting models. These changes are discussed in Section 7.3.3.

The activity of creating models inspired four changes. While creating models, we noticed that it was tedious to initialize variables explicitly as part of a state machine describing the behavior of a class. For this reason, we made it possible to specify the initial value of a variable as part of its declaration. Another design decision that was tedious to deal with while creating models by hand was the existence of a metaclass for signals. In the current version of SLCO, this metaclass is replaced with a simple attribute of type String that denotes the name of a signal. Additionally, the textual syntax for SLCO was introduced because of difficulties encountered while creating models. Besides offering a more convenient way to create and edit models, the textual representation was also a prerequisite for the work presented in Chapter 5. Finally, the distinction between guards, triggers, and statements was removed. Early versions of SLCO distinguish between guards, triggers, and statements, as is the case for the UML. Furthermore, each transition has at most one guard, at most one trigger, and can have any number of statements. Allowing at most one trigger and guard per transition can lead to models with a large number of states. To reduce the number of states, the distinction between guards, triggers, and statements was removed. Figure 3.20 shows the state machines of Figure 7.1 using the new syntax.

Implementing the mappings from SLCO to its target platforms led to two changes. In SLCO, the arguments of all signals sent over a channel must have the same type. A channel in SLCO is characterized by these types, its directionality, its reliability, and its

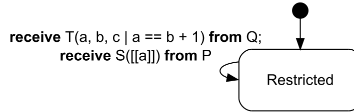



Figure 7.3: Restricted form of conditional signal reception

suitability for either synchronous or asynchronous communication. Initially, the allowed types of the arguments and the directionality of a channel were left implicit. However, the types of the arguments of the signals sent over a channel must be specified explicitly in Promela. When transforming SLCO to Promela, the characteristics of a channel had to be derived from the statements that use the channel for sending and receiving signals. To avoid this, we decided to make all the characteristics of channels explicit, which led to smaller transformations that were easier to understand and modify. Furthermore, while investigating different forms of conditional signal reception and their mapping to the target platforms, we noticed that only some of these forms could be translated to Promela. The signal reception with the condition $a == b + 1$ shown in Figure 7.3, for instance, has no counterpart in Promela that matches the semantics of SLCO. To simplify the implementation of the mapping from SLCO to Promela, we introduced a restricted form of conditional signal reception, which corresponds directly with Promela's construct for conditional signal reception. The statement **receive** $S([[a]])$ **from** $P$ in Figure 7.3, for example, can be translated to Promela straightforwardly.

Defining the mappings from SLCO to the various target platforms and implementing the editors for SLCO models did not lead to any modifications of the language. However, as stated before, additional domain analysis was performed by the platform experts before defining the mappings to the target platforms. The straightforwardness of these mappings is caused by the extension of SLCO that resulted from the additional domain analysis.

### 7.3.3 Evolution of the Transformations

The transformations from SLCO to the various platforms have also evolved. One reason for this evolution was the decision to divide each transformation into steps that either modify existing elements of a model or add elements to a model. We chose this approach because it clarifies the relation between two consecutive steps of a sequence of transformation. It is now easier to see what has changed after applying a single transformation.

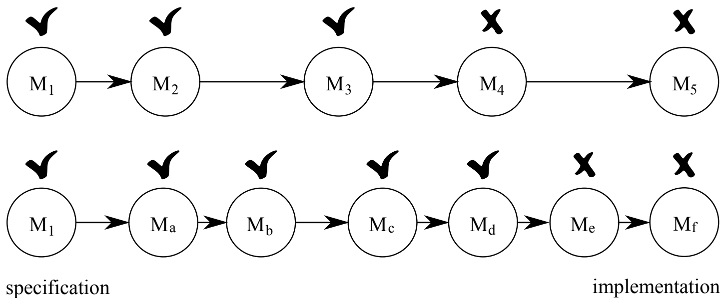



Figure 7.4: A coarse-grained and a fine-grained sequence of transformations

Another reason for splitting up some of the transformations into smaller steps is

to enable verification of intermediate models that are closer to the implementation. This is schematically depicted in Figure 7.4. Before splitting the transformations into smaller parts, models $M_1$, $M_2$, and $M_3$ could be verified using model checking. Verifying model $M_4$ was already impossible due to the state-space explosion problem. After splitting, verification up to model $M_d$ is possible. This model is much closer to the implementation than model $M_3$. Splitting the transformations increased the usability of the models.

### 7.3.4 Generalization

Looking more closely at the evolution of our language and transformations, we noticed that the design choices and changes can be divided into four categories. We consider these categories to characterize the main influences on the evolution of our DSML and transformations.

The choice of defining a language for the specification of systems consisting of concurrent objects is made to effectively describe problems in the *problem domain*. The influence of the problem domain on the evolution of a DSML is to be expected, since the language is focused on that domain.

In our case, the *target platforms* also have an impact on the evolution, since our language does not have its own execution platform, nor is it embedded in another language. If a transformation to, for example, an execution platform is impossible due to language mismatches, it is impossible to execute a model created using the DSML. Therefore, our DSML has evolved to eliminate unresolvable mismatches with the target platforms. The introduction of conditional signal reception and the division of the transformations into smaller steps are enforced by the target platforms.

A large number of design decisions have been made to increase the *quality of models*, in particular their understandability and modifiability. Defining a graphical syntax for the language, adding synchronous communication, introducing local variables, and allowing variables to have an initial value have been done to increase the understandability. Referring to signals based on their name has increased the modifiability of models. The many changes required for increasing the understandability and modifiability of our DSML can be partially explained by our lack of experience in designing DSLs. We found that it is hard to predict which language features will improve understandability and modifiability without actually using the language. These are, however, important quality attributes that should be taken into account to enhance the chance for acceptance of the language in practice. We expect that there are more quality attributes [18] that influence the evolution of a language, which will become apparent when the language is used more extensively.

Enhancing the *quality of the transformations* also had its effect on the language. To increase the understandability and modifiability of the transformations, certain implicit language features have been made explicit, such as the directionality of channels and the argument types of the signals sent over channels.

## 7.4 Related Work

Freeman and Pryce describe their experiences with the evolution of a DSL [41]. Their language has evolved from a library on top of Java to an embedded language. We observed that our DSL has evolved because new target platforms were added. Their language is an embedded language that does not need target platforms. Because the language is embedded, it is highly influenced by the host language, Java in this case. One of the

reasons for the evolution of their language is to improve user-friendliness. This is related to our observation that language evolution is influenced by model quality.

The evolution of a visual programming language for writing real-time control programs for distributed environments and supporting tools is described by Karaila [63]. The main goal of this language is to make programs understandable. The author shows that Lehman's eight laws [75] for characterizing the way software systems tend to evolve are not restricted to software systems, but apply to language evolution as well. The main difference between this language and SLCO is the scale. The language studied by the author has been under development since 1988 and the last major development step dates from 2003. Also the approach of the study is different. Karaila tries to fit his experiences within a framework, whereas we try to distill general lessons from our own experience.

An analysis of over twenty cases where domain-specific languages were designed to implement a model-driven engineering approach is presented by Luoma et al. [77]. The data for the analysis was gathered by means of interviews and discussions with people involved in constructing the DSLs. They identified and categorized four influences on defining DSLs: domain concepts, generated output, look and feel of the system built, and expression of variability. One of the differences with our work is, again, the scope of the study. They studied the definition of over twenty languages, of which some have been used for several years already. We defined only one language, but our focus is on the evolution of the language. Another important difference is that in their case the target platform for the applications developed with the DSL was already chosen, whereas we added multiple target platforms on the fly.

A domain-specific visual language that aims at expressing the evolution of domain-specific visual languages is developed by Sprinkle and Karsai [105]. This language and accompanying tools assist in the co-evolution of domain, DSL and models created using the DSL. They claim that what sets the evolution of domain-specific languages apart is that its primary aim is not backward compatibility but domain specificity. This is in line with our observation that the evolution of a domain-specific language is influenced by the evolution of the problem domain. They focus on tools for supporting DSL evolution, rather than researching the causes for language evolution as we do.

Another framework aimed at supporting DSL evolution is described by De Geest [43]. The primary influence on DSL evolution mentioned by the author is use of the DSL. He claims that by using a DSL, more domain knowledge is gathered, which requires adaptation of the DSL. Also, new features may be requested to ease the modeling process. This is in line with our observation that evolution of a language is influenced by model quality.

Van Deursen and Klint discuss experiences from industrial practice regarding the development of domain-specific languages [31]. The focus of the article is on the development of a DSL for describing financial products. They provide guidelines and considerations that should be taken into account when developing a DSL. Issues as maintainability factors and risks involved in the use of DSLs are also addressed. The DSL they developed has been used for a few years and during those years it has been concluded that the language needed some changes to increase user friendliness. This is in line with our observation that evolution of a language is influenced by model quality.

Karsai et al. provide 26 design guidelines for domain-specific languages [64]. During the development of SLCO, we noticed that the guideline that advocates the reuse of existing language definitions can be in conflict with the guideline that advocates simplicity. Reusing a part of the graphical syntax of the UML and adopting the corresponding distinction between guards, triggers, and statements hampered the understandability of SLCO and complicated the definition of its semantics. Additionally, they advise against

conceptual redundancy, whereas we introduced multiple versions of conditional signal reception to simplify the definition of the semantics and some of the model transformations.

Paige et al. provide 9 principles for the design of modeling languages [88]. The principle of uniqueness is related to the concept of conceptual redundancy mentioned above and also advises to avoid duplication of features. This suggests that conditional signal reception should be revisited during the next redesign of SLCO. Our experience concurs with the provided principles, although seamlessness and reversibility are not applicable for our language. Seamlessness is not applicable because of the variety of target platforms, and reversibility is not applicable because modification of generated code is in conflict with our approach to MDSE. We aim at generating implementations by refining models using model transformations that are provably correct according to some correctness criterion. Modifying the resulting code is not supported, because it could hamper its correctness.

## 7.5 Conclusions and Future Work

This chapter addresses research question $RQ_5$ and identifies four main influences on the evolution of our DSML: the problem domain, the target platforms, model quality, and model transformation quality. The problem domain, model quality, and transformation quality continuously influence the evolution of a language throughout the design process. The problem domain should always be taken into consideration when adapting the language to ensure that the abstractions provided by the language fit the domain. Opportunities to adapt the language in order to improve the quality of models and transformations become apparent as experience with the language grows, while designing and also while using the language. Because quality is a subjective concept, quality attributes can be in conflict. In our case, we added local variables to state machines to increase understandability, which had a negative effect on modifiability.

If the purpose of a DSML changes, transformations to platforms that suit this purpose may be required. However, there may be mismatches between the DSML and the target platform that preclude straightforward transformation. In our experience, the restrictions imposed by the target platform caused the DSML to change in two ways. First, to increase the expressiveness of the language and simplify the definition of its semantics, general forms of missing constructs are added to the language. These general constructs may not have a counterpart on all of the target platforms. Second, to simplify the transformations, additional constructs are added that are less expressive than the aforementioned general constructs, but that have a direct counterpart on all target platforms. The general and restricted form of conditional signal reception form an example of such a change. The general form of conditional signal reception has a counterpart on all platforms except for Spin, and its semantics can be defined straightforwardly. The restricted form of conditional signal reception, however, can be transformed to all platforms, but expressing its semantics is much less straightforward. To facilitate changes like these, SLCO has been divided into two parts. The core of the language is used to concisely define its semantics, whereas the extended version leads to simpler models and transformations.

The main threat to the validity of our research is the scale. The language provides only a limited amount of modeling constructs and we implemented only a limited number of model transformations. Since we find conclusions similar to our own in literature, we expect that our conclusions will also hold for larger scale DSML projects. However, researching the evolution of a more extensive DSML to experience whether different evolution issues arise is relevant future work.

# Chapter 8

## Checking Property Preservation of Refining Transformations

*In model-driven software development, models and model transformations are used to create software. To automatically generate correct software from abstract models by means of model transformations, these transformations must preserve the desirable properties of the initial models. In this chapter, we propose an incremental model checking technique to determine whether model transformations are property preserving. We use labeled transition systems (LTSs) to represent the individual components of models, and formalize model transformations as transformations of LTSs. Checking whether a transformation preserves certain properties involves checking bisimilarity between transformed and new behavior only, instead of comparing the behavior of entire models before and after transformation. Thus, it never requires exploring unchanged behavior twice. We describe our approach and provide experimental results to show its usefulness.*

## 8.1 Introduction

Model-driven software development (MDSD) [17] entails creating implementations on a low level of abstraction from designs represented by models on a high level of abstraction. Implementation details are added to these abstract models by means of refining model transformations. By applying one or more of such transformations to a model that represents the high-level design of a software system, a model on a low level of abstraction can be generated that can be transformed to an implementation for this system straightforwardly. In case a sequence of transformations is applied, also a number of intermediate models is produced. Usually, an implementation must satisfy a number of requirements that can be expressed as properties of the model that forms its design. To ensure that the implementation satisfies its requirements, these properties should also hold for the models that result from model transformations. In other words, these transformations should preserve properties. In this chapter, we address research question $RQ_6$, where a model transformation is considered to be correct if the desirable properties of the input

model are preserved in the refined model. We present a technique to automatically verify the correctness of model transformations.

**RQ$_6$:** *Can we verify the correctness of model transformations automatically?*

Here, we deal with a modeling formalism with a more restricted expressiveness in comparison to the formalism discussed in Chapter 6. Additionally, the model transformations considered below are of a more abstract nature than those presented in Chapter 6.

Traditionally, model checking [26] is applied to verify whether models satisfy certain properties. The models that form the design of a system are often relatively small, which means that properties of such models can easily be verified using traditional techniques such as explicit state-space exploration. Unfortunately, however, iteratively refining these models by adding implementation details quickly results in models that suffer from the state-space explosion problem. This makes it practically unfeasible to perform verification on the resulting refined models by traversing their complete state space. To overcome this problem, we introduce a model checking technique that can be used to determine whether a transformation preserves properties. Using this technique, the fact that the properties that hold for the initial model also hold for the refined model can be inferred from the fact that the transformation is property preserving. In this way, only the initial model needs to be model checked using the traditional technique.

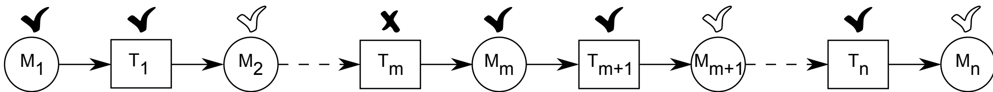



Figure 8.1: Avoiding rechecking intermediate models by checking transformations

The technique presented in this chapter can assert whether a transformation is property preserving for all possible models provided as input. However, if a transformation cannot be shown to preserve properties for all input models, it may still preserve some properties for particular input. In such cases, verification of refined models using traditional techniques may still show that the given properties hold for these models. Figure 8.1 illustrates the application of our technique. It sketches the refinement of model $M_1$, which is transformed to model $M_n$ through a sequence of refinement steps. When a property holds for a model, which is indicated by a black check mark, and this model is transformed using a transformation that preserves this property, which is also indicated by a black check mark, then the property is guaranteed to hold for the resulting model. A white check mark is used to indicate that this is established without rechecking the property for the new model. If a transformation is not guaranteed to preserve a property, the property needs to be rechecked. Transformation $T_m$, for example, does not preserve the property at hand. Therefore, the property needs to be checked again for model $M_m$.

In this chapter, labeled transition systems (LTSs) are used to represent individual processes. By combining these processes with synchronization rules, networks of LTSs [71] are obtained, which represent models. The synchronization rules define how the processes in a network synchronize with each other. A model is refined by transforming its processes using transformation rules and possibly adding additional synchronization rules. Each transformation rule defines how a part of the behavior of a process should be refined. To check whether a refinement step is property preserving, only the transformation rules, a subset of the synchronization rules of the model provided as input, and the newly introduced synchronization rules need to be examined. However, to be able to reason

about property preservation of transformations, a number of reasonable conditions are formulated that must hold for the networks and transformations under consideration. The technique presented here is applicable to any modeling language whose semantics can be expressed in terms of networks of LTSs.

Mateescu and Wijs [78] identified a fragment of the modal $\mu$-calculus [68] that is compatible with divergence-sensitive branching bisimilarity (DSBB) [46]. Because of this compatibility, if a certain property expressed using this fragment of the modal $\mu$-calculus holds for a given model, it also holds for all bisimilar models. The identified fragment is sufficiently expressive to capture the vast majority of practical safety, liveness, and fairness properties. In the same paper, the technique of maximal hiding is introduced. Given a property, maximal hiding enables automatically abstracting away from all behavior that is not relevant for the property. We use both of these results for our preservation check as follows. First, networks of transformation rules are constructed based on the synchronization rules before and after the application of a transformation. Then, the LTSs corresponding to these networks of transformation rules are computed. Each of these networks leads to a pair of LTSs, where one LTS represents the relevant behavior before transformation and the other the behavior after transformation. Finally, property preservation is checked by applying maximal hiding to all the LTSs and checking whether each pair of LTSs is divergence-sensitive branching bisimilar.

The remainder of this chapter is structured as follows. Section 8.2 introduces the preliminaries. In Section 8.3, we formalize LTS transformation. Next, in Section 8.4, we discuss our technique for determining whether transformations preserve properties. Experimental results are given in Section 8.5. Section 8.6 discusses related work, and Section 8.7 contains conclusions and pointers to future work.

## 8.2 Background

In this section, we introduce the basic concepts required to introduce our preservation check. First, finite state LTSs are introduced, which we use to define the behavior of individual processes as well as the behavior of models formed by interacting processes. Then, we discuss networks of LTSs, which are used to specify models in terms of interacting processes. Next, we give a definition of DSBB, an equivalence relation between LTSs. Finally, a fragment of the modal $\mu$-calculus [68] that is compatible with DSBB is discussed.

### 8.2.1 Labeled transition system

An LTS $\mathcal{G}$ is a tuple $\langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G} \rangle$, where $\mathcal{S}_\mathcal{G}$ is the (finite) set of states, $\mathcal{A}_\mathcal{G}$ is the set of actions (including the invisible action $\tau$), $\mathcal{T}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \times \mathcal{S}_\mathcal{G}$ is the transition relation, and $\mathcal{I}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G}$ the set of initial states. Usually, LTSs representing potential behavior of concurrent systems have only one initial state, as defined in Section 6.3.1. Here, we support multiple initial states to enable representing transformation rule patterns in terms of LTSs, as described in Section 8.3. In all other cases, LTSs have a single initial state. Actions in $\mathcal{A}_\mathcal{G}$ are denoted by $a$, $b$, $c$, etc. We use $s_1 \xrightarrow{a}_\mathcal{G} s_2$ as a shorthand for $\langle s_1, a, s_2 \rangle \in \mathcal{T}_\mathcal{G}$. If a transition $s_1 \xrightarrow{a}_\mathcal{G} s_2$ is an element of $\mathcal{T}_\mathcal{G}$, this means that the LTS $\mathcal{G}$ can move from state $s_1$ to state $s_2$ by performing action $a$. The reflexive transitive closure of $\xrightarrow{\tau}_\mathcal{G}$ is denoted by $\Rightarrow_\mathcal{G}$.

### 8.2.2 Network of LTSs

We represent models consisting of a finite number of concurrent processes by a number of LTSs and a set of rules that define how these LTSs interact. For this, we use the concept of networks of LTSs [71]. Given an integer $n > 0$, $1..n$ is the set of integers ranging from 1 to $n$. A vector $\overline{v}$ of size $n$ contains $n$ elements indexed by $1..n$. For $i \in 1..n$, $\overline{v}[i]$ denotes element $i$ in $\overline{v}$.

**Definition 3.** *A network of LTSs $\mathcal{M}$ of size n is a pair $\langle \Pi, \mathcal{V} \rangle$, where*

- *$\Pi$ is vector of n (process) LTSs. For each $i \in 1..n$, we write $\Pi[i] = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i \rangle$, and $s_1 \xrightarrow{b}_i s_2$ is shorthand for $s_1 \xrightarrow{b}_{\Pi[i]} s_2$;*
- *$\mathcal{V}$ is a finite set of synchronization rules. A synchronization rule is a tuple $\langle \bar{t}, a \rangle$, where a is an action label, and $\bar{t}$ is a vector of size n called a synchronization vector, whose elements are action labels from $\bigcup_{i \in 1..n} \mathcal{A}_i$ and a special symbol $\bullet$ that does not occur as a label in the LTSs. If $\bar{t}[i] = \bullet$ for some synchronization rule, then process LTS i is not involved in this synchronization.*

The potential behavior of a network of LTSs can be described as a single LTS. For a network $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$, combining the LTSs in $\Pi$ according to the rules in $\mathcal{V}$ produces a new LTS $\langle \mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, \mathcal{I}_\mathcal{M} \rangle$, where

- $\mathcal{I}_\mathcal{M} = \{ \langle s_1, \ldots, s_n \rangle \mid \forall i \in 1..n . s_i \in \mathcal{I}_i \}$;
- $\mathcal{A}_\mathcal{M} = \{ a \mid \langle \bar{t}, a \rangle \in \mathcal{V} \}$;
- $\mathcal{S}_\mathcal{M} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$;
- $\mathcal{T}_\mathcal{M}$ is the smallest transition relation satisfying:

$$\langle \bar{t}, a \rangle \in \mathcal{V} \wedge \forall i \in 1..n . (\bar{t}[i] = \bullet \wedge s'[i] = s[i]) \vee (\bar{t}[i] \neq \bullet \wedge s[i] \xrightarrow{\bar{t}[i]}_i s'[i]) \implies s \xrightarrow{a}_\mathcal{M} s'.$$

In the remainder, we refer to such an LTS as a network LTS.

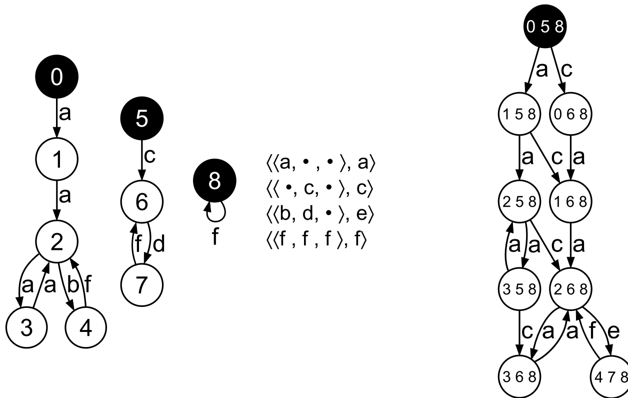



Figure 8.2: A network of LTSs and the corresponding network LTS

On the left of Figure 8.2, a network consisting of three process LTSs and four synchronization rules is shown. The network LTS representing the behavior of this

network is shown on the right of the figure. The figure demonstrates the expressiveness of networks of LTSs. It shows, for example, that multi-party synchronization is offered, as illustrated with the synchronization rule $\langle\langle f, f, f\rangle, f\rangle$. This rule specifies that the action $f$ in the network LTS is the result of the synchronization of the actions $f$ of the three processes. Rule $\langle\langle b, d, \bullet\rangle, e\rangle$ specifies a synchronization between processes $\Pi[1]$ and $\Pi[2]$, rule $\langle\langle a, \bullet, \bullet\rangle, a\rangle$ specifies that action $a$ of process $\Pi[1]$ can be executed independently, and rule $\langle\langle \bullet, c, \bullet\rangle, c\rangle$ specifies the same for action $c$ of process $\Pi[2]$. Synchronization rules can also be used to introduce non-deterministic behavior, by specifying multiple rules involving the same actions. For example, by adding the rule $\langle\langle a, c, \bullet\rangle, g\rangle$ to the network of Figure 8.2, $\Pi[1]$ and $\Pi[2]$ can either synchronize or perform action $a$ and $c$ independently.

To abstract from certain actions, we define the hiding operator $\tau_H$, which renames all actions in $H$ to $\tau$. This operator can be extended to networks of LTSs.

**Definition 4.** *Let $\mathcal{A}$ be a set of actions and $H \subseteq \mathcal{A}$. The hiding of an action $a \in \mathcal{A}$ w.r.t. $H$ is defined as follows.*

$$\tau_H(a) = \begin{cases} a & \text{if } a \notin H \\ \tau & \text{if } a \in H \end{cases}$$

*The hiding of a network $\mathcal{M} = \langle \Pi, \mathcal{V}\rangle$ w.r.t. $H$ is defined as follows.*

$$\tau_H(\langle \Pi, \mathcal{V}\rangle) = \langle \Pi, \{\langle \bar{t}, \tau_H(a)\rangle \mid \langle \bar{t}, a\rangle \in \mathcal{V}\}\rangle$$

### 8.2.3 Divergence-Sensitive Branching Bisimilarity

We use the equivalence relation DSBB [46] to relate LTSs, which preserves $\tau$-cycles and branching-time properties such as inevitable reachability.

**Definition 5.** *For LTSs $\mathcal{G}_1 = \langle \mathcal{S}_{\mathcal{G}_1}, \mathcal{A}_{\mathcal{G}_1}, \mathcal{T}_{\mathcal{G}_1}, \mathcal{I}_{\mathcal{G}_1}\rangle$ and $\mathcal{G}_2 = \langle \mathcal{S}_{\mathcal{G}_2}, \mathcal{A}_{\mathcal{G}_2}, \mathcal{T}_{\mathcal{G}_2}, \mathcal{I}_{\mathcal{G}_2}\rangle$, a binary relation $B \subseteq \mathcal{S}_{\mathcal{G}_1} \times \mathcal{S}_{\mathcal{G}_2}$ is a divergence-sensitive branching bisimulation if for all $s \in \mathcal{S}_{\mathcal{G}_1}$ and $t \in \mathcal{S}_{\mathcal{G}_2}$ such that $s\ B\ t$, the following conditions hold.*

1. *If $s \xrightarrow{a}_{\mathcal{G}_1} s'$ then*
   - *either $a = \tau$ with $s'\ B\ t$;*
   - *or $t \Longrightarrow_{\mathcal{G}_2} \hat{t} \xrightarrow{a}_{\mathcal{G}_2} t'$ with $s\ B\ \hat{t}$ and $s'\ B\ t'$.*
2. *Symmetrically, if $t \xrightarrow{a}_{\mathcal{G}_2} t'$ then*
   - *either $a = \tau$ with $s\ B\ t'$;*
   - *or $s \Longrightarrow_{\mathcal{G}_1} \hat{s} \xrightarrow{a}_{\mathcal{G}_1} s'$ with $\hat{s}\ B\ t$ and $s'\ B\ t'$.*
3. *If for all $k \geq 0$ and $s = s_0$, $s_k \xrightarrow{\tau}_{\mathcal{G}_1} s_{k+1}$ then for all $\ell \geq 0$ and $t = t_0$, $t_\ell \xrightarrow{\tau}_{\mathcal{G}_2} t_{\ell+1}$ and $s_k\ B\ t_\ell$, for all $k, \ell$.*
4. *Symmetrically, if for all $k \geq 0$ and $t = t_0$, $t_k \xrightarrow{\tau}_{\mathcal{G}_2} t_{k+1}$ then for all $\ell \geq 0$ and $s = s_0$, $s_\ell \xrightarrow{\tau}_{\mathcal{G}_1} s_{\ell+1}$ and $s_\ell\ B\ t_k$, for all $k, \ell$.*

*Two states $s$ and $t$ are divergence-sensitive branching bisimilar, denoted by $s \leftrightarrow_b^{\Delta} t$, if there is a divergence-sensitive branching bisimulation $B$ with $s\ B\ t$.*

Two LTSs $\mathcal{G}_1 = \langle \mathcal{S}_{\mathcal{G}_1}, \mathcal{A}_{\mathcal{G}_1}, \mathcal{T}_{\mathcal{G}_1}, \mathcal{I}_{\mathcal{G}_1} \rangle$ and $\mathcal{G}_2 = \langle \mathcal{S}_{\mathcal{G}_2}, \mathcal{A}_{\mathcal{G}_2}, \mathcal{T}_{\mathcal{G}_2}, \mathcal{I}_{\mathcal{G}_2} \rangle$ are divergence-sensitive branching bisimilar, denoted by $\mathcal{G}_1 \leftrightarrow_b^\Delta \mathcal{G}_2$, iff $\forall s_1 \in \mathcal{I}_{\mathcal{G}_1}.\exists s_2 \in \mathcal{I}_{\mathcal{G}_2}.s_1 \leftrightarrow_b^\Delta s_2$ and vice versa. Furthermore, a state $s$ is diverging, denoted by $s\uparrow$, iff an infinite $\tau$-path is reachable from $s$. For finite LTSs, this means that a $\tau$-cycle is reachable via $\tau$-transitions.

If the constituting LTSs satisfy the following conditions related to $\tau$-transitions, DSBB is a congruence for networks of LTSs, which means that replacing a process LTS in a network with a process LTS that is divergence-sensitive branching bisimilar leads to a network for which the corresponding network LTS is divergence-sensitive branching bisimilar with the network LTS that corresponds to the original network.

**Definition 6.** *A network $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ is called admissible iff the following holds.*

1. ***No synchronization and renaming:***

$$\forall \langle \bar{t}, a \rangle \in \mathcal{V}.\bar{t}[i] = \tau \implies Ac(\bar{t}) = \{i\} \wedge a = \tau;$$

2. ***No cut:*** $\exists s_1, s_2 \in \mathcal{S}_i.s_1 \xrightarrow{\tau}_i s_2 \implies \exists \langle \bar{t}, \tau \rangle \in \mathcal{V}.\bar{t}[i] = \tau,$

where the set $Ac(\bar{t})$ of indices of processes active for a synchronization vector $\bar{t}$ is defined as $Ac(\bar{t}) = \{i \mid i \in 1..n \wedge \bar{t}[i] \neq \bullet\}$. The first condition states that a $\tau$-action may not synchronize with other actions and that it may not be renamed, and the second condition states that there must exist synchronization rules that enable the $\tau$-transitions of each process. In the remainder of this chapter, only admissible networks are considered.

### 8.2.4 The modal $\mu$-calculus $L_\mu^{dsbr}$

Mateescu and Wijs [78] identified a fragment of the modal $\mu$-calculus [68] that is fully compatible with the maximal hiding technique [78] and DSBB. The fragment is called $L_\mu^{dsbr}$ and is suitable for expressing safety, liveness, and fairness properties. In $L_\mu^{dsbr}$, properties are expressed in terms of action formulas, denoted by $\alpha$, and state formulas, denoted by $\varphi$ and $\psi$. The syntax of these formulas and the semantics of the action formulas are defined as follows.

$$\alpha \quad ::= \quad a \mid \mathsf{false} \mid \neg\alpha_1 \mid \alpha_1 \vee \alpha_2$$

$$\begin{aligned} [\![a]\!]_\mathcal{A} &= \{a\} \\ [\![\mathsf{false}]\!]_\mathcal{A} &= \emptyset \\ [\![\neg\alpha_1]\!]_\mathcal{A} &= \mathcal{A} \setminus [\![\alpha_1]\!]_\mathcal{A} \\ [\![\alpha_1 \vee \alpha_2]\!]_\mathcal{A} &= [\![\alpha_1]\!]_\mathcal{A} \cup [\![\alpha_2]\!]_\mathcal{A} \end{aligned}$$

$$\begin{aligned} \varphi \quad &::= \quad \mathsf{false} \mid \neg\varphi_1 \mid \varphi_1 \vee \varphi_2 \mid \langle(\varphi_1?.\alpha_h)^*\rangle\psi \mid \langle\varphi_1?.\alpha_h\rangle@ \mid X \mid \mu X.\varphi_1 \\ \psi \quad &::= \quad \varphi \mid \langle\alpha_v\rangle\varphi \mid \neg\psi_1 \mid \psi_1 \vee \psi_2, \end{aligned}$$

where $\mathcal{A}$ is a set of actions, $X$ is a set of propositional variables, $\tau \in [\![\alpha_h]\!]_\mathcal{A}$, and $\tau \notin [\![\alpha_v]\!]_\mathcal{A}$.

Interpretation $[\![\alpha]\!]_\mathcal{A}$ of $\alpha$ on the set of actions $\mathcal{A}$ denotes the set of actions satisfying $\alpha$. An action $a$ satisfies a formula $\alpha$, denoted by $a \models_\mathcal{A} \alpha$, iff $a \in [\![\alpha]\!]_\mathcal{A}$. Furthermore, a transition $s_1 \xrightarrow{a}_\mathcal{G} s_2$ with $a \models_{\mathcal{A}_\mathcal{G}} \alpha$ is called an $\alpha$-transition.

We do not provide the formal semantics of the state formulas, as it is not essential for understanding the current work. Compared to the standard modal $\mu$-calculus, the fragment $L_\mu^{dsbr}$ differs in two respects. First, it introduces the weak possibility modality and the weak infinite looping operator. Informally, the weak possibility modality $\langle(\varphi_1?.\alpha_h)^*\rangle\psi$

characterizes the states having an outgoing sequence of zero or more $\alpha_h$-transitions whose intermediate states satisfy $\varphi_1$ and whose terminal state satisfies $\psi$. The weak infinite looping operator $\langle \varphi_1 ?.\alpha_h \rangle @$ characterizes the states having an infinite outgoing sequence of $\alpha_h$-transitions whose intermediate states satisfy $\varphi_1$. In both cases, $\alpha_h$ must capture $\tau$. Second, the occurrence of the strong modality $\langle \alpha_v \rangle \varphi$ is restricted syntactically such that it can appear only after a weak possibility modality, and the action formula $\alpha_v$ must denote visible actions only. For further details, we refer to the work of Mateescu and Wijs [78].

The fact that a state $s$ of an LTS $\mathcal{G}$ satisfies a closed state formula $\varphi$ is denoted by $s \models_{\mathcal{G}} \varphi$. A formula is closed if all propositional variables are bound. Additionally, we denote $\forall s \in \mathcal{I}_{\mathcal{G}}.s \models_{\mathcal{G}} \varphi$ by $\models_{\mathcal{G}} \varphi$.

When checking a state formula $\varphi$ on an LTS, some actions can be hidden (renamed to $\tau$) without disturbing the interpretation of $\varphi$.

**Definition 7.** *Let $\alpha$ be an action formula interpreted over a set of actions $\mathcal{A}$. The hiding set of $\alpha$ w.r.t. $\mathcal{A}$ is defined as follows.*

$$h_{\mathcal{A}}(\alpha) = \begin{cases} [\![\alpha]\!]_{\mathcal{A}} & \text{if } \tau \models \alpha \\ \mathcal{A} \setminus [\![\alpha]\!]_{\mathcal{A}} & \text{if } \tau \not\models \alpha \end{cases}$$

*The hiding set of a state formula $\varphi$ w.r.t. $\mathcal{A}$, denoted by $h_{\mathcal{A}}(\varphi)$, is defined as the intersection of $h_{\mathcal{A}}(\alpha)$ for all action subformulas $\alpha$ of $\varphi$.*

We denote maximal hiding in an LTS $\mathcal{G} = \langle \mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}} \rangle$ as $\tilde{\tau}_{\varphi}(\mathcal{G}) = \tau_{h_{\mathcal{A}_{\mathcal{G}}}(\varphi)}(\mathcal{G})$. Maximal hiding preserves $L_{\mu}^{dsbr}$ properties: $\models_{\mathcal{G}} \varphi \iff \models_{\tilde{\tau}_{\varphi}(\mathcal{G})} \varphi$. Furthermore, for closed $\varphi$, $L_{\mu}^{dsbr}$ is compatible with the $\underline{\leftrightarrow}_b^{\Delta}$ relation. Let $\mathcal{G} = \langle \mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}} \rangle$ be an LTS and let $s_1, s_2 \in \mathcal{S}_{\mathcal{G}}$ such that $s_1 \underline{\leftrightarrow}_b^{\Delta} s_2$. Then, $s_1 \models_{\mathcal{G}} \varphi \iff s_2 \models_{\mathcal{G}} \varphi$ for any closed state formula $\varphi$ of $L_{\mu}^{dsbr}$. This allows reducing an LTS (after maximal hiding) modulo DSBB before verifying a closed $L_{\mu}^{dsbr}$ formula [78]. It also allows reasoning about LTSs w.r.t. properties. Given a $L_{\mu}^{dsbr}$ formula $\varphi$ and LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ with $\tilde{\tau}_{\varphi}(\mathcal{G}_1) \underline{\leftrightarrow}_b^{\Delta} \tilde{\tau}_{\varphi}(\mathcal{G}_2)$, then, if $\models_{\mathcal{G}_1} \varphi$, also $\models_{\mathcal{G}_2} \varphi$.

## 8.3 LTS Transformations

In this section, we formalize refinement steps as transformations of networks of LTSs. A network is transformed by transforming the individual process LTSs that constitute it and adding additional synchronization rules.

### 8.3.1 Transformation Rules

LTSs are transformed by applying transformation rules. These rules are defined as follows.

**Definition 8.** *An LTS transformation rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ consists of a left pattern LTS $\mathcal{L}^r = \langle \mathcal{S}_{\mathcal{L}^r}, \mathcal{A}_{\mathcal{L}^r}, \mathcal{T}_{\mathcal{L}^r}, \mathcal{I}_{\mathcal{L}^r} \rangle$ and a right pattern LTS $\mathcal{R}^r = \langle \mathcal{S}_{\mathcal{R}^r}, \mathcal{A}_{\mathcal{R}^r}, \mathcal{T}_{\mathcal{R}^r}, \mathcal{I}_{\mathcal{R}^r} \rangle$, with $\mathcal{I}_{\mathcal{L}^r} = \mathcal{I}_{\mathcal{R}^r} = (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r})$.*

States $\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}$, also referred to as the glue-states, are all initial and define how $\mathcal{R}^r$ should replace $\mathcal{L}^r$. All changes to an LTS are applied relative to these glue-states. We call a rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ applicable on an LTS $\mathcal{G}$ iff there exists a match $m_r : \mathcal{S}_{\mathcal{L}^r} \to \mathcal{S}_{\mathcal{G}}$ for which the following holds.

**Definition 9.** *A transformation rule* $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *has a match* $m_r : \mathcal{S}_{\mathcal{L}^r} \to \mathcal{S}_{\mathcal{G}}$ *on an LTS* $\mathcal{G} = \langle \mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}} \rangle$ *iff* $m_r$ *is injective and*

1. $\forall s_1 \xrightarrow{a}_{\mathcal{L}^r} s_2 . m_r(s_1) \xrightarrow{a}_{\mathcal{G}} m_r(s_2)$;
2. $\forall s_1 \in \mathcal{S}_{\mathcal{L}^r} \setminus \mathcal{S}_{\mathcal{R}^r}, s_2 \in \mathcal{S}_{\mathcal{G}}$ :
   - $m_r(s_1) \xrightarrow{a}_{\mathcal{G}} s_2 \implies \exists s \in \mathcal{S}_{\mathcal{L}^r} . s_1 \xrightarrow{a} s \wedge m_r(s) = s_2$;
   - $s_2 \xrightarrow{a}_{\mathcal{G}} m_r(s_1) \implies \exists s \in \mathcal{S}_{\mathcal{L}^r} . s \xrightarrow{a} s_1 \wedge m_r(s) = s_2$.

The second condition of Definition 9 is related to what are often called dangling edges. In graph transformation, dangling edges (transitions) are usually removed as part of a transformation. Here, we decide to make a rule non-applicable in case dangling transitions are present. Otherwise, the effect of a transformation would be hard to predict based only on the rule itself, because it may cause states that are not present in the rule to become unreachable.

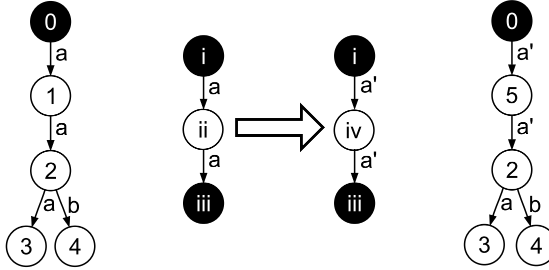



Figure 8.3: Transformation rule matching

In the middle of Figure 8.3, a transformation rule is shown. All initial and glue-states are colored black in this figure. The rule defines that any state matched on state *ii* of the left pattern of the rule should be removed and replaced by a new state, which is labeled *iv* in the rule. Therefore, the left pattern can be matched on states $\{0, 1, 2\}$ of the LTS on the left of the figure, but not on states $\{1, 2, 3\}$. The latter match would result in the removal of state 2 and lead to a dangling transition.

If a rule $r$ is applicable to an LTS $\mathcal{G}$, then $\mathcal{G}$ can be transformed as follows.

**Definition 10.** *The transformation of an LTS* $\mathcal{G} = \langle \mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}} \rangle$ *according to a rule* $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *and a given match* $m_r : \mathcal{S}_{\mathcal{L}^r} \to \mathcal{S}_{\mathcal{G}}$ *is defined as follows.* $T_r^{m_r}(\mathcal{G}) = \langle \mathcal{S}_r^{m_r}, \mathcal{A}_r^{m_r}, \mathcal{T}_r^{m_r}, \mathcal{I}_{\mathcal{G}} \rangle$, *where*

- $\mathcal{S}_r^{m_r} = (\mathcal{S}_{\mathcal{G}} \setminus \{m_r(s) \mid s \in (\mathcal{S}_{\mathcal{L}^r} \setminus \mathcal{S}_{\mathcal{R}^r})\}) \cup (\mathcal{S}_{\mathcal{R}^r} \setminus \mathcal{S}_{\mathcal{L}^r})$;
- $\mathcal{T}_r^{m_r} = (\mathcal{T}_{\mathcal{G}} \setminus \{\langle m_r(s_1), a, m_r(s_2) \rangle \mid s_1 \xrightarrow{a}_{\mathcal{L}^r} s_2\}) \cup$
  $\{\langle s_1, a, s_2 \rangle \mid s_1, s_2 \in (\mathcal{S}_{\mathcal{R}^r} \setminus \mathcal{S}_{\mathcal{L}^r}) \wedge s_1 \xrightarrow{a}_{\mathcal{R}^r} s_2\} \cup$
  $\{\langle m_r(s_1), a, s_2 \rangle \mid s_1 \in (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}) \wedge s_2 \in (\mathcal{S}_{\mathcal{R}^r} \setminus \mathcal{S}_{\mathcal{L}^r}) \wedge s_1 \xrightarrow{a}_{\mathcal{R}^r} s_2\} \cup$
  $\{\langle s_1, a, m_r(s_2) \rangle \mid s_1 \in (\mathcal{S}_{\mathcal{R}^r} \setminus \mathcal{S}_{\mathcal{L}^r}) \wedge s_2 \in (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}) \wedge s_1 \xrightarrow{a}_{\mathcal{R}^r} s_2\} \cup$
  $\{\langle m_r(s_1), a, m_r(s_2) \rangle \mid s_1, s_2 \in (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}) \wedge s_1 \xrightarrow{a}_{\mathcal{R}^r} s_2\}$;
- $\mathcal{A}_r^{m_r} = (\mathcal{A}_{\mathcal{G}} \setminus \mathcal{A}_{\mathcal{L}^r}) \cup \mathcal{A}_{\mathcal{R}^r} \cup \{\tau\}$.

The new set of states $\mathcal{S}_r^{m_r}$ consists of $\mathcal{S}_\mathcal{G}$ without the states that correspond to the states in the left pattern that do not exist in the right pattern, the removed states, and with the states in the right pattern that do not exist in the left pattern, the newly added states. We assume that the latter states are fresh in $\mathcal{S}_r^{m_r}$. Furthermore, transitions $\mathcal{T}_r^{m_r}$ consist of $\mathcal{T}_\mathcal{G}$ without the transitions that correspond to the transitions in the left pattern and with the transitions that correspond to those in the right pattern.

Figure 8.3 illustrates the application of a transformation rule. The LTS on the right of the figure is the result of applying the rule in the middle to the LTS on the left.

### 8.3.2 Rule Systems

With transformation rules, a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ can be built, with $R$ a set of transformation rules and $\hat{\mathcal{V}}$ a set of synchronization rules. Contrary to related work on graph transformation, rule systems are not used to describe the semantics of a system in our setting. Instead, they define how a network of LTSs is transformed into a more refined network. Therefore, we are not interested in all possible interleavings of applications of the rules in a rule system. Let $\mathcal{G} \Rrightarrow_R \mathcal{G}'$ denote the fact that an LTS $\mathcal{G}'$ can be obtained by applying a rule $r \in R$ on one match in LTS $\mathcal{G}$, and let $\Rrightarrow_R^*$ be the reflexive, transitive closure of $\Rrightarrow_R$. Then, $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ is terminating iff $\Rrightarrow_R$ is terminating, and $\Sigma$ is confluent iff the following holds.

**Definition 11.** *Let $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ be a rule system and $\mathcal{G}$ be an LTS. $\Sigma$ is confluent iff for all LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ with $\mathcal{G} \Rrightarrow_R^* \mathcal{G}_1$ and $\mathcal{G} \Rrightarrow_R^* \mathcal{G}_2$, there exists an LTS $\mathcal{G}_3$ such that $\mathcal{G}_1 \Rrightarrow_R^* \mathcal{G}_3$ and $\mathcal{G}_2 \Rrightarrow_R^* \mathcal{G}_3$*

In the remainder of this chapter, we assume that rule systems are terminating and confluent. From graph theory, it is known that confluence is undecidable for general rule systems, but it is decidable under certain conditions [70, 94]. Here, we ensure that a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ is terminating and confluent for an LTS $\mathcal{G}$ by requiring that the following two conditions hold.

1. **No new matches:** $\forall r \in R. \mathcal{A}_{\mathcal{R}^r} \cap \bigcup_{r' \in R} \mathcal{A}_{\mathcal{L}^{r'}} = \emptyset$;
2. **Remove single match:** $\bigcap_{r \in R} \mathcal{A}_{\mathcal{L}^r} = \emptyset$,

The first condition ensures that the application of a transformation rule does not introduce new matches, by requiring that all actions in the right-hand patterns of the rules do not occur in any of the left-hand patterns. The second condition ensures that exactly one match is removed, by requiring that none of the left-hand patterns contains an action that also occurs in another left-hand pattern. Both conditions can be checked straightforwardly by inspecting the rule system only.

For terminating, confluent $\Sigma$, applying all $r \in R$ as often as possible results in a particular LTS, independent of the order of rule application. We refer to that LTS as $T_R^+(\mathcal{G})$. Finally, we define the transformation of a network of LTSs.

**Definition 12.** *Given a network $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ and a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$, the transformed network is defined as follows, for $n = |\Pi|$.*

$$T_\Sigma(\mathcal{M}) = \langle \langle T_R^+(\Pi[1]), \ldots, T_R^+(\Pi[n]) \rangle, \mathcal{V} \cup \hat{\mathcal{V}} \rangle$$

## 8.4 Checking Property Preservation

In this section, we show how property preservation of transformations can be checked by generating networks from rule systems and comparing the LTSs of these networks. Figure 8.4 gives an overview of the approach.

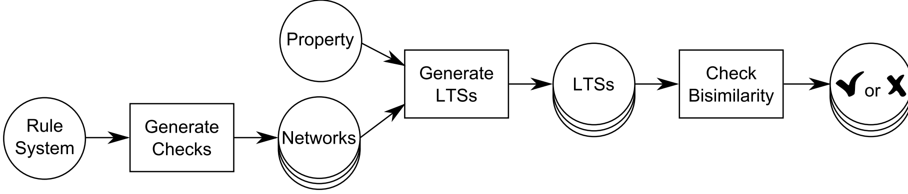



Figure 8.4: Checking property preservation by comparing LTSs

First, networks are generated based on the left and right-hand sides of transformation rules. Then, the network LTSs corresponding to these networks are generated, while applying maximal hiding regarding the property at hand. Finally, property preservation is checked by comparing the network LTSs generated from left-hand sides of transformation rules with the network LTSs of the corresponding right-hand sides. If all pairs of LTSs are divergence-sensitive branching bisimilar, the rule system preserves the property at hand.

More formally, a terminating, confluent rule system $\Sigma$ is $\varphi$-preserving for a property $\varphi \in L_\mu^{dsbr}$ iff $\models_\mathcal{M} \varphi \iff \models_{T_\Sigma(\mathcal{M})} \varphi$ for all networks $\mathcal{M}$. Thus, if $\Sigma$ is $\varphi$-preserving and $\models_\mathcal{M} \varphi$, we can conclude that $\models_{T_\Sigma(\mathcal{M})} \varphi$ without rechecking property $\varphi$ for network $T_\Sigma(\mathcal{M})$. Since $L_\mu^{dsbr}$ is compatible with maximal hiding and DSBB, as discussed in Section 8.2.4, $\Sigma$ is $\varphi$-preserving if $\tilde{\tau}_\varphi(\mathcal{M}) \leftrightarroweq_b^\Delta \tilde{\tau}_\varphi(T_\Sigma(\mathcal{M}))$. In this section, we discuss under which conditions a rule system implies the bisimilarity of $\tilde{\tau}_\varphi(\mathcal{M})$ and $\tilde{\tau}_\varphi(T_\Sigma(\mathcal{M}))$. The most important condition roughly boils down to checking whether, after some appropriate rewriting, the left and right patterns of the transformation rules are divergence-sensitive branching bisimilar after maximal hiding. If this is the case, then applying the rules does not result in a network with structurally different behavior.

Without loss of generality, for each network $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ and rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$, we assume that each $r \in R$ has exactly one match to some $\Pi[i]$ and that each $\Pi[i]$ is matched on by exactly one $r$. This is expressed by indexing the $r \in R$ such that rule $r_i$ is matched on $\Pi[i]$. If $R$ contains only one rule, we omit its index. Since $\Sigma$ is confluent, the results of this section can be lifted to the more general case where rules may have an arbitrary number of matches. With this assumption, it can also safely be assumed that all the $\mathcal{A}_i$ are disjoint. If this is not the case, some renaming of actions and a corresponding modification of the synchronization rules can resolve this.

### 8.4.1 Extended Transformation Rules

To show that a rule system $\Sigma$ preserves a property $\varphi$ for every network $\mathcal{M}$, we show that a divergence-sensitive branching bisimulation between the states of $\tilde{\tau}_\varphi(\mathcal{M})$ and $\tilde{\tau}_\varphi(T_\Sigma(\mathcal{M}))$ can be constructed based on the divergence-sensitive branching bisimulations between networks constructed from the transformation rules. To construct these networks from the transformation rules, we extend the transformation rules with self-loops on the glue-states. The matches of the glue-states of a given transformation rule may be part of transitions that are not present in the patterns of this rule. The transformation rules are extended

to make explicit that such transitions may exist. Each self-loop is labeled with an action uniquely related to the corresponding state.

**Definition 13.** *Given an LTS transformation rule* $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$*, the corresponding extended transformation rule is defined as* $r_\kappa = \langle \mathcal{L}^r_\kappa, \mathcal{R}^r_\kappa \rangle$*, where*

- $\mathcal{L}^r_\kappa = \langle \mathcal{S}_{\mathcal{L}^r}, \mathcal{A}_{\mathcal{L}^r} \cup \{\kappa_s \mid s \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}\}, \mathcal{T}_{\mathcal{L}^r} \cup \{\langle s, \kappa_s, s \rangle \mid s \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}\}, \mathcal{I}_{\mathcal{L}^r} \rangle$*;*
- $\mathcal{R}^r_\kappa = \langle \mathcal{S}_{\mathcal{R}^r}, \mathcal{A}_{\mathcal{R}^r} \cup \{\kappa_s \mid s \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}\}, \mathcal{T}_{\mathcal{R}^r} \cup \{\langle s, \kappa_s, s \rangle \mid s \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}\}, \mathcal{I}_{\mathcal{R}^r} \rangle$*.*

We assume that the $\kappa$-actions are not originally in $\mathcal{A}_{\mathcal{L}^r}$. Without these loops, a DSBB check of patterns could consider two deadlock states to be bisimilar, while they are actually different glue-states that are possibly matched on states with different outgoing transitions not present in the patterns. For example, without the dashed $\kappa$-loops, states *ii* and *iii* in Figure 8.5 would be related if $a, b \in h_{\mathcal{A}}(\varphi)$. With the $\kappa$-loops, however, they are not related, as indicated by the cross. Thus, the extra transitions ensure that $\mathcal{L}^r_\kappa \leftrightarrow^{\Delta}_{b} \mathcal{R}^r_\kappa$ iff there exists a divergence-sensitive branching bisimulation that relates all the glue-states to themselves. A glue-state $s$ in $\mathcal{L}^r_\kappa$ (or $\mathcal{R}^r_\kappa$) with self-loop $s \xrightarrow{\kappa_s} s$ must at least be related to itself in $\mathcal{R}^r_\kappa$ (or $\mathcal{L}^r_\kappa$) since it is the only state where a $\kappa_s$-transition is enabled.

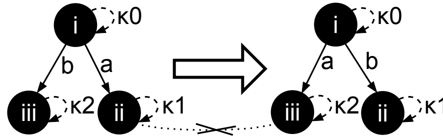



Figure 8.5: An extended transformation rule

### 8.4.2 Synchronizing Behavior

If a rule system consists of multiple transformation rules, then multiple LTS transformations can be applied in a single transformation step. To check $\varphi$-preservation of such rule systems, we need to take possible synchronization between different rule patterns into account.

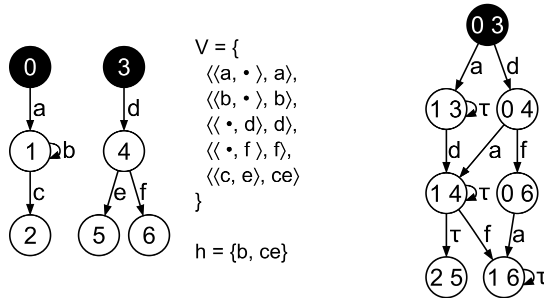



Figure 8.6: A network and the corresponding network LTS

In Figure 8.6, a network involving synchronization and the corresponding network LTS are shown. The network LTS is obtained after hiding the actions in $h$, shown in the middle of the figure, which is the hiding set for some property $\varphi$. The rule system shown on the left of Figure 8.7 can be applied to the network of Figure 8.6. Individually, the

rules seem to fundamentally change the behavior of the process LTSs, as shown in the middle of Figure 8.7. However, since the rule system also adds the new synchronization rules $\langle\langle c1, e1\rangle, c1e1\rangle$ and $\langle\langle c2, \bullet\rangle, c2\rangle$, and the actions *c1e1* and *c2* can be hidden, the final network LTS, as shown on the right of the figure, is bisimilar to the one before transformation. To incorporate such possible dependencies between rule patterns, we developed a $\varphi$-preservation check involving networks of rule patterns.

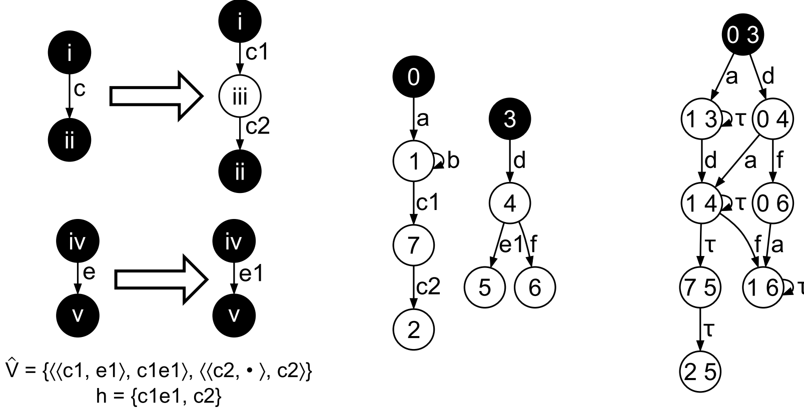



Figure 8.7: A rule system and an example of its application

In general, when considering transformation rules that affect synchronizing actions and thus involve multiple process LTSs, it cannot be determined whether a given rule system $\Sigma = \langle R, \hat{\mathcal{V}}\rangle$ involving such rules is $\varphi$-preserving by just analyzing the $\mathcal{L}^{r_i}$ and $\mathcal{R}^{r_i}$ of all $r_i \in R$. However, this can be done if $\Sigma$ has a number of properties regarding synchronizing behavior of a network $\mathcal{M}$, which we together call synchronization uniformity.

Before we can give a definition of synchronization uniformity, we need a number of auxiliary definitions. The set of actions involved in synchronization vector $\bar{t}$ is $\mathcal{A}(\bar{t}) = \{a \mid \exists i \in 1..n.\bar{t}[i] = a \wedge a \neq \bullet\}$, and the set of actions involved in synchronization rules in $\mathcal{V}$ with multiple processes is $\mathcal{A}_s(\mathcal{V}) = \{a \mid \exists\langle\bar{t}, a'\rangle \in \mathcal{V}.a \in \mathcal{A}(\bar{t}) \wedge |Ac(\bar{t})| > 1\}$. The set of indices of process LTSs that can potentially synchronize with behavior in $\mathcal{L}^{r_i}$ according to a synchronization rule in $\mathcal{V}$ is $dep(\mathcal{L}^{r_i}, \mathcal{V}) = \bigcup\{Ac(\bar{t}) \mid \langle\bar{t}, a\rangle \in \mathcal{V} \wedge \bar{t}[i] \in \mathcal{A}_{\mathcal{L}^{r_i}}\}$. This definition states that $j$ is in $dep(\mathcal{L}^{r_i}, \mathcal{V})$ iff there exists a synchronization rule $\langle\bar{t}, a\rangle$ in $\mathcal{V}$ such that both $i \neq \bullet$ and $j \neq \bullet$, i.e. both $i$ and $j$ are active for that rule, and the behavior in $\Pi[i]$ is matched on by transformation rule $r_i = \langle\mathcal{L}^{r_i}, \mathcal{R}^{r_i}\rangle$. The set of actions of process $j$ on which the actions in $\mathcal{L}^{r_i}$ depend according to the synchronization rules in $\mathcal{V}$ is $A^{\mathcal{L}^{r_i}}_{dep}(j, \mathcal{V}) = \{\bar{t}[j] \mid \langle\bar{t}, a\rangle \in \mathcal{V} \wedge \bar{t}[i] \in \mathcal{A}_{\mathcal{L}^{r_i}} \wedge \bar{t}[j] \neq \bullet\}$. In other words, the set of all actions $\bigcup_{t \in F}\{\bar{t}[j]\} \setminus \{\bullet\}$ constitutes $A^{\mathcal{L}^{r_i}}_{dep}(j, \mathcal{V})$, where $F$ represents the set of all synchronization rules applicable on $\mathcal{L}^{r_i}$. Sets $dep(\mathcal{R}^{r_i}, \mathcal{V})$ and $A^{\mathcal{R}^{r_i}}_{dep}(j, \mathcal{V})$ are defined similarly. Now, synchronization uniformity can be defined.

**Definition 14.** *We say that rule system $\Sigma = \langle R, \hat{\mathcal{V}}\rangle$ is synchronization uniform w.r.t. network $\mathcal{M} = \langle\Pi, \mathcal{V}\rangle$ iff the following holds.*

1. $\forall a \in \mathcal{A}_s(\mathcal{V}).(\exists r_i \in R.a \in \mathcal{A}_{\mathcal{L}^{r_i}}) \implies \forall s_1 \xrightarrow{a}_i s_2.s_1, s_2 \in m_{r_i}(\mathcal{S}_{\mathcal{L}^{r_i}})$;
2. $\forall r_i \in R, j \in dep(\mathcal{L}^{r_i}, \mathcal{V}).A^{\mathcal{L}^{r_i}}_{dep}(j, \mathcal{V}) \subseteq \mathcal{A}_{\mathcal{L}^{r_j}}$;
3. $\forall\langle\bar{t}, a\rangle \in \hat{\mathcal{V}}, i \in 1..n.\bar{t}[i] = \bullet \vee \bar{t}[i] \in \mathcal{A}_{\mathcal{R}^{r_i}}$.

The first condition states that if a transformation rule is applicable to a synchronizing transition, then it is applicable to all synchronizing transitions with the same label in $\mathcal{M}$. If this is not guaranteed, it becomes very hard to reason about the model after transformation because it is difficult to determine a priori exactly which transitions in different process LTSs will be able to synchronize in the network. Therefore, predicting the effect of rewriting, for example, $a$-transitions in some places while keeping other $a$-transitions the same is as difficult. Checking this condition requires inspecting the process LTSs of a network, unless we impose an additional restriction on rule systems. If we require that all left-hand patterns of rules that modify synchronizing transitions consist of a single transition, the first condition holds, regardless of the structure of the process LTSs of $\mathcal{M}$. The second condition states that all actions that can synchronize with $\mathcal{L}^{r_i}$ are also transformed by $\Sigma$. If this does not hold, it becomes hard to analyze the synchronizing behavior as appearing in transformation patterns. In such cases, some of the behavior that is relevant for this analysis is not present in any of the patterns, which makes analysis based only on the rule system impossible. Finally, the third condition states that each new synchronization rule $\langle \overline{t}, a\rangle \in \hat{\mathcal{V}}$ involves actions from the $\mathcal{R}^{r_i}$ of the corresponding rule $r_i$ only. It is crucial to rule out the possibility of transforming merely by introducing synchronization rules, because this also prevents analysis solely based on rule systems. For example, if we define a new synchronization rule involving existing actions $a$ and $b$, and these actions were previously not allowed to synchronize, then we clearly change the model without actually transforming anything.

In the remainder of this chapter, we only consider rule systems that are synchronization uniform regarding a given model. This may seem a big assumption, but in practice, one tends to transform synchronizing behavior in a uniform way. Usually, synchronizing actions, say $a$ and $b$, represent communication. If one wants to transform this behavior, it is natural to do this consistently in all places where $a$ and $b$ occur, and to transform the behavior of both communicating parties to keep them compatible with each other.

### 8.4.3 Networks of transformation rules

From a rule system, networks of transformation rules can be constructed.

**Definition 15.** *For a model $\mathcal{M} = \langle \Pi, \mathcal{V}\rangle$, rule system $\Sigma = \langle R, \hat{\mathcal{V}}\rangle$, and a rule $r_i \in R$, the vector $\xi^{r_i}$ of transformation rules relevant for the behavior in $\mathcal{L}^{r_i}$ is defined as follows, for all $j \in 1..n$.*

$$\xi^{r_i}[j] = \begin{cases} * \mapsto * & \text{if } j \notin dep(\mathcal{L}^{r_i}, \mathcal{V}) \\ r_{j,\kappa} & \text{if } j \in dep(\mathcal{L}^{r_i}, \mathcal{V}), \end{cases}$$

where $*$ is a dummy state. For a given vector $\xi^{r_i}$, $\xi^{r_i}_{\mathcal{L}}$ is the vector of left patterns of the extended transformation rules in $\xi^{r_i}$, and $\xi^{r_i}_{\mathcal{R}}$ is the vector of right patterns. The networks $\Xi^{r_i}_{\mathcal{L}} = (\xi^{r_i}_{\mathcal{L}}, \mathcal{V})$ and $\Xi^{r_i}_{\mathcal{R}} = (\xi^{r_i}_{\mathcal{R}}, \mathcal{V} \cup \hat{\mathcal{V}})$ allow comparing synchronizing behavior in rule patterns, before and after transformation according to $\Sigma$, in particular involving $r_i$.

On the left of Figure 8.8, another example of a rule system is shown. In general, this rule system is not $\varphi$-preserving. Applying this rule system to the network in Figure 8.6 results in the network shown in the middle of Figure 8.8. The corresponding network LTS shown on the right of this figure is obtained after hiding the actions in $h$. The networks of the left and right patterns of the two transformation rules in this figure are shown in Figure 8.9b. Actions in $h$ are hidden. The dotted lines in this figure illustrate that a divergence-sensitive branching bisimulation exists for these two networks.

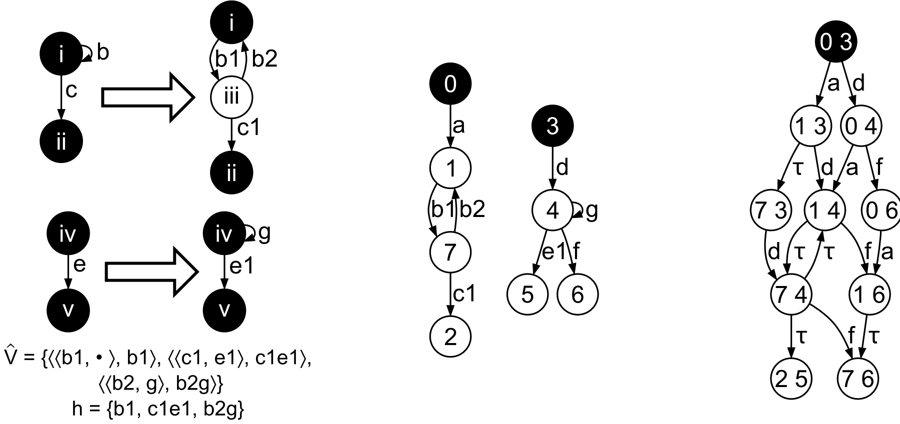
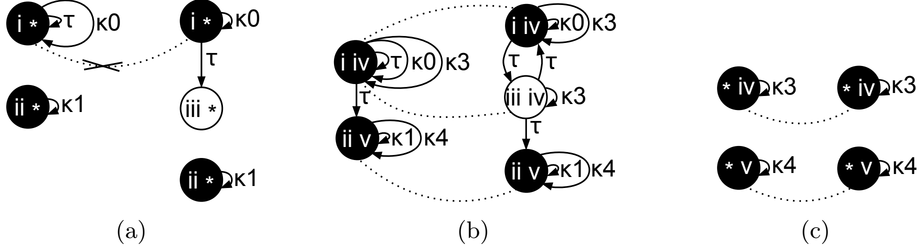



Figure 8.8: A non-preserving transformation and an example of its application




(a) (b) (c)

Figure 8.9: Network LTSs of networks of transformation rules

Even though a divergence-sensitive branching bisimulation exists between the networks of the left and right patterns of the transformation rules in Figure 8.8, the rule system is not $\varphi$-preserving. This clearly shows that it is not sufficient to only take successful synchronization in account. Instead, we also need to consider situations in which some parties are able to perform a synchronizing action whereas at least one other party involved in the synchronization is not. For example, the state labeled (1 3) in Figure 8.6 has a $\tau$-loop that cannot be simulated by the network LTS in Figure 8.8. This $\tau$-loop is the result of hiding the $b$-loop of the leftmost process of Figure 8.6. This process can perform action $b$ independently. After transformation, however, a $\tau$-cycle can only result from interaction between the transformed process LTSs shown in the middle of Figure 8.8. In situations where both processes are able to interact successfully, an infinite number of $\tau$-actions can be performed, as shown in Figure 8.9b. However, if the required synchronization between processes is impossible, as is the case in the state labeled (1 3) in Figure 8.8, only one $\tau$-action can be performed.

To be able to consider such scenarios, we define a projection operator on networks of transformation rules.

**Definition 16.** *For each vector of transformation rules $\xi^{r_i}$ and $j \in 1..n$, the projection operator $/I$ $(I \subseteq 1..n)$ is defined as follows.*

$$\xi^{r_i}/I[j] = \begin{cases} \xi^{r_i}[j] & \text{if } j \in I \\ * \mapsto * & \text{otherwise} \end{cases}$$

This operator can similarly be applied on the vectors of patterns $\xi_{\mathcal{L}}^{r_i}$ and $\xi_{\mathcal{R}}^{r_i}$, and we say that $\Xi_{\mathcal{L}}^{r_i}/I = (\xi_{\mathcal{L}}^{r_i}/I, \mathcal{V})$ and $\Xi_{\mathcal{R}}^{r_i}/I = (\xi_{\mathcal{R}}^{r_i}/I, \mathcal{V} \cup \hat{\mathcal{V}})$, for a given model $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ and rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ such that $r_i \in R$.

Figure 8.9a shows the left and right patterns of the rule network that contains only the topmost transformation rule of Figure 8.8. The left pattern of this rule network is constructed from the left pattern of this transformation rule, using the synchronization rules and the hiding set of Figure 8.6. The right pattern of this rule network is constructed from the right pattern of this transformation rule, using the synchronization rules and the hiding set of Figure 8.8. Because these patterns are not divergence-sensitive branching bisimilar, as indicated by the cross, this rule system is not $\varphi$-preserving. The remaining patterns are shown in Figure 8.9c.

### 8.4.4 Constructing a Bisimulation Relation

The following theorem formalizes our $\varphi$-preservation check.

**Theorem 2.** *Let $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ be a model and $\varphi \in L_{\mu}^{dsbr}$ a temporal property such that $\models_{\mathcal{M}} \varphi$. Then $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ is $\varphi$-preserving if the following holds, for all $r_i \in R$, $I \subseteq dep(\mathcal{L}^{r_i}, \mathcal{V})$.*

$$\tilde{\tau}_{\varphi}(\Xi_{\mathcal{L}}^{r_i}/I) \leftrightarrow_b^{\Delta} \tilde{\tau}_{\varphi}(\Xi_{\mathcal{R}}^{r_i}/I) \tag{8.1}$$

Proving this theorem for a rule system $\Sigma$ and a property $\varphi$ entails constructing a relation between $\tilde{\tau}_{\varphi}(\mathcal{M})$ and $\tilde{\tau}_{\varphi}(T_{\Sigma}(\mathcal{M}))$ based on the divergence-sensitive branching bisimulations in equation 8.1, for some model $\mathcal{M}$, and proving that this relation is a divergence-sensitive branching bisimulation too. To describe the construction of this relation, we need a number of auxiliary definitions.

First, a state in the network LTS of network $\mathcal{M}$ is a vector $\bar{s} = \langle \bar{s}[1], \ldots, \bar{s}[n] \rangle$. An arbitrary $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ can have up to $n$ elements that are matched on by some transformation rule. For an LTS $\mathcal{G}$ and a transformation rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$, matching $m_r$ is extended to sets of states such that $m_r(\mathcal{S}_{\mathcal{L}^r}) \subseteq \mathcal{S}_{\mathcal{G}}$ refers to the set of states to which the states in $\mathcal{S}_{\mathcal{L}^r}$ are matched, and $\hat{m}_r(\mathcal{S}_{\mathcal{L}^r}) = \{s \in m_r(\mathcal{S}_{\mathcal{L}^r}) \mid m_r^{-1}(s) \notin \mathcal{S}_{\mathcal{R}^r}\}$ refers to the states in $\mathcal{S}_{\mathcal{G}}$ that relate to non-glue-states in $\mathcal{L}^r$. By definition, the latter states are those that are removed from $\mathcal{G}$ by rule $r$. We denote the set of indices of elements in $\bar{s}$ matched on by the corresponding rule with $M(\bar{s}) = \{i \mid \bar{s}[i] \in m_{r_i}(\mathcal{S}_{\mathcal{L}^{r_i}})\}$. As indicated above, we assume that if $\bar{s}[i]$ is matched on, then it is matched on by rule $r_i$. Furthermore, for a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ and a network $\mathcal{M}$, a simulation relation exists between the states in $\mathcal{M}$ and those in the rule networks $\Xi_{\mathcal{L}}^{r_i}/I$ as well as between the states in $T_{\Sigma}(\mathcal{M})$ and those in $\Xi_{\mathcal{R}}^{r_i}/I$. This can be formalized as follows. For a state vector $\bar{s}^*$ in a rule network $\Xi_{\mathcal{L}}^{r_i}/I$ and a state vector $\bar{s}$ in an LTS network $\mathcal{M}$, we say that $\bar{s}$ simulates $\bar{s}^*$, denoted $\bar{s}^* \vdash \bar{s}$, iff $\forall i \in 1..n. \bar{s}^*[i] \neq * \implies \bar{s}[i] = m_{r_i}(\bar{s}^*[i])$. In other words, besides the $*$-states, all process states in $\bar{s}^*$ are matched on the corresponding states in $\bar{s}$. Now, first of all, if $\bar{s}^* \vdash \bar{s}$, and $\bar{s}^* \xrightarrow{a} \bar{s}^{*\prime}$, then also $\bar{s} \xrightarrow{a} \bar{s}'$ and $\bar{s}^{*\prime} \vdash \bar{s}'$. Second of all, in cases that $\bar{s} \xrightarrow{a} \bar{s}'$ and some synchronization vector $\bar{t}$ enables transition $\bar{s} \xrightarrow{a} \bar{s}'$, with $Ac(\bar{t}) \subseteq I$, $Ac(\bar{t}) \subseteq M(\bar{s})$, and $Ac(\bar{t}) \subseteq M(\bar{s}')$, then the involved behavior of every active $\Pi[i]$ ($i \in Ac(\bar{t})$) is matched on by $r_i$. The definition for $\Xi_{\mathcal{R}}^{r_i}/I$ and $T_{\Sigma}(\mathcal{M})$ is similar. These simulation relations are preserved after maximal hiding.

A relation between two networks representing a model before and after transformation can be constructed by combining the bisimulations between pairs of rule networks with the simulation relations between model networks and rule networks. In Figure 8.10, the dashed lines connect some of the states in the LTSs of the rule networks with the

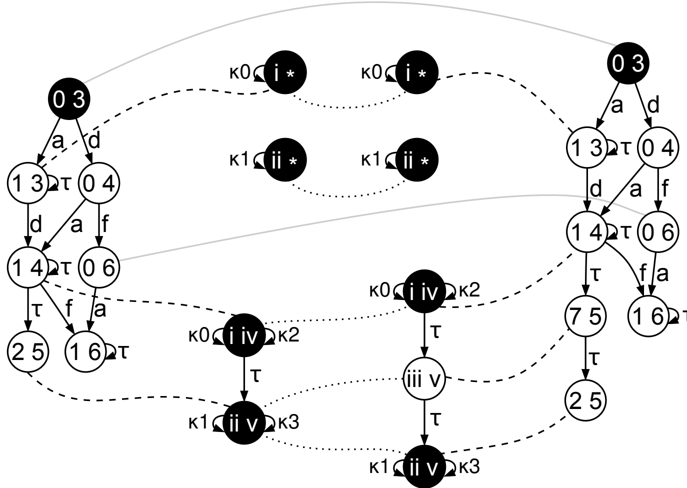



Figure 8.10: Constructing a divergence-sensitive branching bisimulation

states in the LTSs of the models that simulate them. The dotted lines in the figure denote the divergence-sensitive branching bisimulations between pairs of rule networks. By combining these relations, states in the LTS on the left of the figure are related to states in the LTS on the right. We refer to the relation formed by combining these relations as $D$. To increase the readability of the figure, not all states that are related according to $D$ are connected. For states $\overline{s}$ of $\mathcal{M}$ and $\overline{p}$ of $T_\Sigma(\mathcal{M})$, we define $\overline{s}\ D'\ \overline{p}$ iff $\forall i \notin M(\overline{s}).\overline{s}[i] = \overline{p}[i]$. In words, $D'$ relates all state vectors with exactly the same elements apart from those matched on by a transformation rule. In a sense, $D'$ is a strong bisimulation for the behavior not subjected to transformation. As mentioned above, not all states that are related according to $D$ are connected in the figure. Only those states of the LTS on the left of the figure are connected to states of the LTS on the right that are also related according to $D'$. Thus, the dashed and dotted lines together illustrate the relation $(D \cap D')$ by connecting some of the states that are related according to this relation. Relation $D'' = \{(\overline{s}, \overline{s}) \mid M(\overline{s}) = \emptyset\}$ relates all states that are not subjected to transformation. The grey lines in Figure 8.10 connect the states that are related according to $D''$. Given these relations, a relation $C = (D \cap D') \cup D''$ can be constructed, which relates $\tilde{\tau}_\varphi(\mathcal{M})$ and $\tilde{\tau}_\varphi(T_\Sigma(\mathcal{M}))$. By considering the cases of Definition 5, it can be shown that this relation is a divergence-sensitive branching bisimulation [40].

## 8.5 Experimental Results

Table 8.1 shows experimental results for five case studies with various rule systems. The number of explored states and the runtime for full exploration are given for the initial model and the transformed model. The applied rule systems have been analyzed using the proposed technique. For the resulting checks, the maximum number of states of the two LTSs involved in a check is given in the form "(size left pattern)+(size right pattern)". If a rule system is property preserving, this is denoted by ✓ in the table. The result of an unsuccessful preservation check is denoted by ✗. Furthermore, the number of required checks and the total runtime are reported. We report the time needed to perform a

| | | ACS | 1394-fin | wafer | broadcast | | ABP | |
|---|---|---|---|---|---|---|---|---|
| **initial model** | *#states* | 4,764 | 188,569 | 78,919 | 161,051 | 161,051 | 759,375 | 759,375 |
| | *time* (sec.) | 1.85 | 379.08 | 4.88 | 3.48 | 3.48 | 29.97 | 29.97 |
| **transformed model** | *#states* | 21,936 | 5,849,124 | 375,937 | 4,084,101 | 28,629,151 | 656,356,768 | 656,356,768 |
| | *time* (sec.) | 10.23 | 18,045.13 | 49.33 | 83.53 | 952.85 | 48,795.28 | 45,553.27 |
| **preservation checks** | *max. #st.* | 2+3 | 2+6 | 2+5 | 27+30 | 27+31 | 15+58 | 15+58 |
| | *#checks* | 1 | 1 | 1 | 7 | 7 | 63 | 63 |
| | *result* | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ |
| | *time* (sec.) | 0.01 | 0.01 | 0.01 | 0.616 | 0.792 | 1.90 | 1.90 |

Table 8.1: Experimental results

complete state-space exploration because this provides an indication of the amount of time required to (re)check any given property. The experiments have been performed on a machine with two dual-core AMD OPTERON (tm) processors 885 2.6 GHz, 126 GB RAM, running RED HAT 4.3.2-7. For DSBB checking, we used the ltscompare tool of the mCRL2 toolset [49]. The results clearly show that checking property preservation takes much less time than exploration of the state space of transformed models.

The first three models are part of the distribution of mCRL2. We generated their LTSs with the mCRL2 tools and manually transformed them to incorporate refined information concerning internal steps. The applied transformation is similar to the one shown in Figure 8.7. In the other two cases, synchronizing behavior was transformed, and the network LTSs have been constructed from sets of process LTSs using Exp.Open [71]. The case study named *broadcast* concerns a system of five sets of three processes communicating via broadcast. The three processes in each set synchronize simultaneously. The rule systems we applied break this down into a series of two-party synchronizations. We defined two rule systems for this, and they could be applied fifty times using a proprietary prototype tool. The first of these rule systems does not preserve properties, and the second one does. The case study named *ABP* concerns a system in which communication between two processes is refined to use the alternating bit protocol [12] in five different places. The rule networks for the various checks were produced by our prototype. We analyzed two versions of the rule system, one containing the subtle error that the receiver of messages does not expect messages with the wrong bit. A more detailed description of the case studies is available in Appendix E.

## 8.6 Related Work

The work presented in this chapter is related to incremental model checking. Early papers on this subject propose techniques to reuse model checking results of safety properties for a given LTS to determine whether it still satisfies the same property after some alterations [104, 107]. This work focuses on particular models, whereas our technique deals with property preservation of transformations in general. Large speedups are reported compared to complete rechecking, but the memory requirements are at least as high, since all states plus additional bookkeeping per state must reside in memory. Our technique does not require such bookkeeping. Furthermore, we do not deal with large, flat LTSs directly, but with networks and transformation rules that both consist of relatively small LTSs. Finally, we do not recheck a property after transformation, but check bisimilarity instead.

The work described in this chapter is also related to action refinement [48], which provides a way to describe and study the top-down design of concurrent systems. Action refinement allows specifying how a more concrete description of a concurrent system can

be obtained from an abstract description of this system by replacing certain actions by more detailed behavior. Action refinement deals with replacing single actions, whereas our work deals with replacing patterns that represent more involved behavior.

Saha presents an incremental algorithm for updating bisimulation relations based on changes of a graph that is related to our work, although it is used in a different context [100]. The goal of this work is efficiently maintaining a bisimulation, whereas the goal of our work essentially is to assess whether a bisimulation exists.

Combemale et al. [27], Hülsbusch et al. [57], and Karsai and Narayanan [65, 83] check semantics preservation of model transformations using either strong or weak bisimilarity. They consider exogenous, horizontal transformations [79], which transform models from one language to another without changing their level of abstraction. In contrast, our work deals with endogenous, vertical transformations, which have the same input and output language, and change the level of abstraction of models. The approach of Hülsbusch et al. and our approach are transformation-dependent and input-independent [2], whereas the work of Combemale et al. and the work of Karsai and Narayanan is transformation-dependent and input-dependent.

Giese et al. relate input and output models when specifying a transformation and use a theorem prover to show semantic equivalence between the input and output of the transformation [44]. A downside of this approach is that it is not completely automated and thus requires manual labor, whereas our approach is automated. Schätz verifies the preservation of properties of a structural nature for model transformations [101], also using a theorem prover. Both techniques are transformation-dependent and model-independent, and deal with horizontal transformations. Giese et al. consider exogenous transformations, and Schätz endogenous ones.

## 8.7 Conclusions and Future Work

In this chapter, we addressed research question $RQ_6$ for restricted forms of models and model transformations. We presented a technique to check whether refining model transformations preserve properties. It is aimed at verifying the correctness of complex models that are the result of iterative refinement through model transformation. Models are formally represented by networks of LTSs and model transformations as rule systems. We can check whether specific safety, liveness, and fairness properties are preserved by rule systems that are terminating, confluent, and synchronization uniform. If a rule system preserves a property that holds for a given input model, construction and exploration of the LTS of a model obtained by transformation can be avoided. If, however, the proposed technique cannot establish that a transformation preserves a property for all models, it is still possible that this property is preserved for certain input models. In such cases, traditional techniques for model checking can be employed to verify the property for the transformed model. Checking multiple properties simply involves performing the required checks for multiple hiding sets. Experiments have shown that checking whether a transformation preserves a given property outperforms rechecking the property for transformed models.

There are two main directions for future research, which are aimed at extending the presented approach to more expressive formalism for the description of models and model transformations. First, the concept of networks of LTSs could be extended to support additional features such as asynchronous communication and variables for storing information. Second, a more expressive formalism to describe model transformations

could be introduced. Currently, for example, it is not possible to express the addition or removal of processes, which is also not yet supported by the technique for checking property preservation. By extending the expressiveness of the formalisms used to describe models and model transformations, and extending the technique for checking property preservation correspondingly, an automated alternative for manual correctness proofs such as those described in Chapter 6 could be obtained.

# Chapter 9

# Conclusions

*This chapter concludes this thesis by discussing the main contributions and directions for future research. For each of the research questions stated in Chapter 1, we provide the main results and conclusions. Additional details are available in the chapters that cover the research questions.*

## 9.1 Contributions

The main research question covered in this thesis is formulated as follows.

> **RQ:** *How can we improve the reliability of software that is automatically generated from high-level descriptions?*

This question is divided into six more specific research questions, and each of these questions is addressed in one of the chapters of this thesis.

The first of these questions deals with the efficient creation of models that form high-level descriptions of software systems and is formulated as follows.

> **$\mathbf{RQ}_1$:** *How can large models for existing modeling languages be created efficiently using existing tools?*

To address this question, we investigated two approaches for the integration of textual and graphical modeling languages, as described in Chapter 2. As a case study, we implemented a textual surface language as an alternative for the activity diagrams of the UML. Tools that integrate this textual language with the UML have been implemented using grammarware and modelware. The main advantage of the approach that uses grammarware is the flexibility it offers while defining the grammar of the surface language. However, because models containing fragments of surface language are transformed to plain models by rewriting the XMI representations of these models, the main disadvantage of this approach is its low level of abstraction. In contrast, the approach that uses modelware poses more restrictions on the grammar of the surface language, but deals with

concepts related to models directly, instead of their XMI representations. A case study showed that this surface language provides a convenient way of creating large, detailed UML models. By replacing activity diagrams with fragments of surface language, the number of diagrams used to describe all aspects of a given system could be significantly reduced, without hampering the understandability of the resulting model.

Part of the work described in this thesis is aimed at generating software from models on a high level of abstraction by refining these models. Each refinement step adds implementation details to a given model, and after applying a sequence of such transformations, a model is obtained that is sufficiently detailed to be transformed to an implementation. This refinement leads to a number of intermediate models. Research question $\mathrm{RQ}_2$ addresses the relation between the amount of detail added by model transformations and the verifiability of intermediate models using traditional model checking techniques.

> $\mathbf{RQ_2}$**:** *How does the size and complexity of model transformations affect the verifiability of intermediate models produced by sequences of refining model transformations?*

The research described in Chapter 4 compares iterative refinement of models using coarse-grained and fine-grained sequences of model transformations. In comparison to coarse-grained sequences of transformations, fine-grained sequences of transformations add less implementation details in each refinement step. Because the models that form the end result of such sequences must contain the same amount of detail to be able to transform them to an implementation, fine-grained sequences of transformations lead to a larger number of intermediate models. Therefore, the difference in abstraction level between each pair of intermediate models produced by a sequence of transformations is smaller if this sequence is fine-grained. Experiments showed that adding implementation details to models with sequences of transformations that are too coarse-grained often leads to models that suffer from state-space explosion, even when these sequences are applied to small source models. When iteratively refining models with such coarse-grained sequences of transformations, the intermediate models obtained after a few refinement steps are too complex to be verified using explicit state-space exploration. As a result, the models for which verification using traditional model checking techniques can be applied describe systems on a much higher level of abstraction than that of the implementation of these systems. With fine-grained sequences of transformations, it is possible to apply model checking to models that are closer to the implementation. Additional advantages of fine-grained sequences of transformations are that the transformations used to form such sequences are more reusable than those used to form coarse-grained sequences and that the size of these transformations makes it easier to locate defects. Their improved composability makes it possible to combine these transformations into multiple sequences leading to implementations with different characteristics based on the same input model.

To verify the correctness of the refining model transformations for SLCO, a formal semantics of the language is required. Before defining the formal semantics, we implemented an executable prototype to experiment with various design decisions, which is described in Chapter 5. The following research question is related to this prototype.

> $\mathbf{RQ_3}$**:** *What are the advantages and disadvantages of implementing an executable prototype of the semantics of a domain-specific modeling language using ASF+SDF?*

We implemented a number of tools that together produce the state space of a given SLCO model. Connecting these tools to existing tools for verification and visualization enabled

experimenting with different variants of the semantics and facilitated the development of model transformations. The biggest advantage of using the ASF+SDF Meta-Environment for the implementation of these tools is its ability to automatically generate command-line tools that offer fast execution of rewrite rules and efficient use of memory. Because state spaces of models describing the behavior of concurrent systems are often very large, generating them quickly using as little memory as possible is important. Although the ASF+SDF Meta-Environment is still available and can be used without problems on computers with a 32-bit architecture, its development has stopped. The fact that 64-bit architectures are not supported precludes using over 4 GB of memory, which is a disadvantage for memory-intensive applications such as state-space generation.

To produce reliable software from models by means of model transformations, these transformations must preserve certain desirable properties of the models. Research question $RQ_4$ addresses this issue.

> **$RQ_4$:** *Can we show that the model transformations that we implemented to refine SLCO models preserve certain desirable properties of such models?*

In Chapter 6, we describe a formal framework for reasoning about the correctness of the endogenous model transformations related to SLCO. This framework relies on the formal semantics of SLCO, based on the previously mentioned prototype, and branching bisimilarity. According to this framework, a transformation is considered to be correct if the observable behavior of any input model is equivalent to the observable behavior of the corresponding output model, after appropriate renaming and hiding of actions. By developing sequences of transformations that are as fine-grained as possible, the number of straightforward correctness proofs was increased, and the number of proof obligations for the correctness proofs of larger transformations was reduced.

Research question $RQ_5$ is concerned with the evolution of SLCO. Over time, various target platforms were added and the way of defining the semantics of the language changed. These changes in turn triggered other changes to the languages and its transformations. To learn from our experiences and to be able to apply the lessons learned while developing other DSMLs, Chapter 7 addresses the following research question.

> **$RQ_5$:** *What are the main influences on the design of a DSML and the corresponding model transformations?*

We identified four main influences on the evolution of our DSML: the problem domain, the target platforms, model quality, and model transformation quality. The language and its transformations continuously changed as a result of adding new target platforms and improving the quality of models and model transformations. In some cases, changes that improve certain aspects of the language have a negative influence on other aspects. To reduce the number of such conflicts, SLCO has been divided into two parts. The core of the language facilitates the concise definition of its semantics, and the extended version, whose semantics can be expressed in terms of constructs of the core language, facilitates the creation of simple models and model transformations.

Verifying the correctness of models transformations as described in Chapter 6 requires a significant amount of manual labor. Therefore, we formulated the following research question.

> **$RQ_6$:** *Can we verify the correctness of model transformations automatically?*

In Chapter 8, a model transformation is considered to be correct if it preserves certain desirable properties for all models provided as input. To automatically verify whether a transformation is property preserving, we propose a technique that performs a number of divergence-sensitive branching bisimilarity checks on labeled transition systems created from the input and output patterns of transformation rules. Models are formalized as networks of labeled transition systems, and model transformations are formalized as rule systems containing the aforementioned transformation rules. If a rule system is property preserving, then each model generated by applying this rule system to a given input model satisfies the same properties as the input model. If a rule system does not preserve a certain property for all models in general, however, it may still produce a valid output model when applied to particular input models. In such cases, traditional techniques for model checking can be employed to check the resulting models.

The techniques and approaches discussed in this thesis enable the generation of reliable and correct software from concise, formal models specified on a high level of abstraction. We developed a DSML with an intuitive graphical syntax for the creation of such models, and a number of model transformations for the automated generation of executable code from these models. As stated in Chapter 1, early validation of model transformations is equally important as early validation of models when producing software using model transformations. For the validation of models, we presented transformations to existing formalisms for simulation and verification, and a custom tool for the generation of state spaces. For the validation of model transformations, we presented two approaches to determine whether these transformations are correct in the sense that they preserve certain properties. By combining these techniques and approaches, we showed that an implementation for a given system can be generated automatically from a design of this system in the form of a model that satisfies certain desirable properties by applying model transformations that preserve these properties.

## 9.2 Future Work

The integration of surface language fragments into the UML as described in Chapter 2 showed to be a useful way of reducing the number of graphical diagrams describing trivial behavior without hampering the understandability of the involved models. However, both of the approaches we describe that enable this integration have their disadvantages. An interesting direction for future research would be to investigate whether the flexibility offered by the language definition formalisms of the grammarware tools can be achieved in the context of modelware.

We identified a number of differences between SLCO and its target languages, as discussed in Chapters 3 and 4. For each of these differences, we developed one or more endogenous model transformations, which are used to bridge the gaps between SLCO and its target languages. However, there are additional differences between the languages, which could be addressed by additional refining model transformations. For instance, the number of state machines describing the behavior of the instances of a single class is unlimited in SLCO, whereas NQC offers a limited amount of tasks per controller. Because each state machine in SLCO is transformed into a task in NQC, this discrepancy could lead to problems. Fortunately, these problems could be prevented with a transformation that merges multiple state machines into a single state machine. This transformation might also be reused by the transformation that merges objects, to replace the protocol that is currently employed to simulate synchronous communication between state machines by

means of shared variables. This change would reduce the number of possible interleavings of actions performed by the objects, which in turn improves the verifiability of intermediate models as discussed in Chapter 4.

In Chapter 6, we mention that a large number of the correctness proofs for the transformations related to SLCO are relatively straightforward. Therefore, the application of automated theorem proving to the work presented in this chapter is considered a promising direction for future research. Additionally, successful termination and time are not yet taken into account. Finding an appropriate equivalence relation that matches the notion of time discussed in Chapter 5 and extending the formal semantics are two of the challenges regarding the latter extension. Finally, we chose to extend SLCO with a number of features related to asynchronous communication over unreliable channels, to be able to refine models using endogenous transformations only, and to be able to limit ourselves to dealing with the formal semantics of one language only for the correctness proofs of these transformations. Alternatively, an approach could be taken that involves a larger number of languages, where each of these languages is a variant of SLCO that differs slightly from the others. In that way, a variant of SLCO in which objects can only communicate synchronously would be transformed into an implementation in NQC via a number of intermediate languages. In each of these languages, a particular language feature is replaced by another feature, such that each consecutive language variant has more features in common with the implementation language than the previous variant. For this approach, a more modular way of defining semantics is needed [59, 82], to avoid repetition in the definitions of the semantics of these language variants. This is also considered a valuable direction for future work.

The technique to check property preservation of model transformations discussed in Chapter 8 can be extended in a number of ways. For example, the concept of networks of LTSs could be extended to support features offered by modeling languages such as SLCO. Currently, networks of LTSs offer no direct support for time, data, and asynchronous communication. The model transformations described in Chapter 3 have been implemented in Xtend and ATL. The relation between languages used in practice for the implementation of model transformations, such as these two, and the formal notion of rule systems needs further study. Some of the changes applied to models by the transformations of Chapter 3 cannot be expressed in terms of a rule system. For instance, adding and removing processes is not supported. Furthermore, the proposed technique for checking property preservation is also not able to deal with this type of modification. Therefore, enabling support for adding and removing processes is another valuable direction for future research.

# Bibliography


[1] J.-R. Abrial, M.K.O. Lee, D. Neilson, P.N. Scharbach, and I.H. Sørensen. The B-Method. In *Proceedings of the 4th International Symposium of VDM Europe*, 1991. doi:10.1007/BFb0020001.

[2] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le Traon, and J.R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012. doi:10.1109/ICST.2012.197.

[3] M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. PhD thesis, Eindhoven University of Technology, 2011.

[4] M.F. van Amstel, S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. In Vitro Design of a Domain-Specific Modeling Language. Submitted to Science of Computer Programming, 2012.

[5] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. An Exercise in Iterative Domain-Specific Language Design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution*, 2010. doi:10.1145/1862372.1862386.

[6] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. Using a DSL and Fine-Grained Model Transformations to Explore the Boundaries of Model Verification. In *Proceedings of the 3rd Workshop on Model-Based Verification and Validation*, 2011. doi:10.1109/SSIRI-C.2011.26.

[7] M.F. van Amstel, M.G.J. van den Brand, Z. Protić, and T. Verhoeff. Model-Driven Software Engineering. In *Automation in Warehouse Development*. Springer, 2011. doi:10.1007/978-0-85729-968-0_4.

[8] M.F. van Amstel, M.G.J. van den Brand, Z. Protić, and T. Verhoeff. Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap? In *Proceedings of the 1st International Conference on Model Transformation*, 2008. doi:10.1007/978-3-540-69927-9_5.

[9] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Prototyping the Semantics of a DSL using ASF+SDF: Link to Formal Verification of DSL Models. In *Proceedings of the 2nd International Workshop on Algebraic Methods in Model-based Software Engineering*, 2011. doi:10.4204/EPTCS.56.5.

[10] S. Andova, M.G.J. van den Brand, and L.J.P. Engelen. Reusable and Correct Endogenous Model Transformations. In *Proceedings of the 5th International Conference on Model Transformation*, 2012. doi:10.1007/978-3-642-30476-7_5.

[11] J. Baeten and C.A. Middelburg. *Process Algebra with Timing*. Springer, 2002.

[12] K.A. Bartlett, R.A. Scantlebury, and P.T. Wilkinson. A Note on Reliable Full-Duplex Transmission over Half-Duplex Links. *Communications of the ACM*, 1969. doi:10.1145/362946.362970.

[13] D. Baum. *NQC Programmer's Guide*, 2003. http://bricxcc.sourceforge.net/nqc/doc/.

[14] D.A. van Beek, P. Collins, D.E. Nadales Agut, J.E. Rooda, and R.R.H. Schiffelers. New Concepts in the Abstract Format of the Compositional Interchange Format. In *Proceedings of the 3rd IFAC Conference on Analysis and Design of Hybrid Systems*, 2009. doi:10.3182/20090916-3-ES-3003.00044.

[15] D.A. van Beek, A.T. Hofkamp, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and Formal Semantics of Chi 2.0. Technical report, Department of Mechanical Engineering, Eindhoven University of Technology, 2008.

[16] J.A. Bergstra. *Algebraic Specification*, chapter 1. ACM, 1989.

[17] S. Beydeda, M. Book, and V. Gruhn. *Model-Driven Software Development*. Springer, 2005.

[18] B.W. Boehm, J.R. Brown, H. Kaspar, M. Lipow, G.J. Macleod, and M.J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[19] M.G.J. van den Brand, L.J.P. Engelen, M. Hamilton, A. Levytskyy, and J.P.M. Voeten. Embedded Systems Modeling, Analysis and Synthesis. In *Ideals: Evolvability of Software-Intensive High-Tech Systems*. Embedded Systems Institute, 2007.

[20] M.G.J. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of the 10th International Conference on Compiler Construction*, 2001. doi:10.1007/3-540-45306-7_26.

[21] M.G.J. van den Brand and P. Klint. ATerms for Manipulation and Exchange of Structured Data: It's All About Sharing. *Information and Software Technology*, 2007. doi:10.1016/j.infsof.2006.08.009.

[22] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 2008. doi:http://dx.doi.org/10.1016/j.scico.2007.11.003.

[23] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 1998. doi:10.1109/32.708566.

[24] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-Evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, 2008. doi:10.1109/EDOC.2008.44.

[25] E.M. Clarke, Jr., O. Grumberg, and D.E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 1994. doi:10.1145/186025.186051.

[26] E.M. Clarke, Jr., O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[27] B. Combemale, X. Crégut, P.-L. Garoche, and X. Thirioux. Essay On Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software*, 2009. doi:10.4304/jsw.4.9.943-958.

[28] J.R. Cordy. The TXL Source Transformation Language. *Science of Computer Programming*, 2006. doi:10.1016/j.scico.2006.04.002.

[29] R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM*, 1995. doi:10.1145/201019.201032.

[30] A. van Deursen. An overview of ASF+SDF. In *Language Prototyping: An Algebraic Specification Approach*. World Scientific Publishing Co., 1996.

[31] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 1998. doi:10.1002/(SICI)1096-908X(199803/04)10:2<75::AID-SMR168>3.0.CO;2-5.

[32] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 2000. doi:10.1145/352029.352035.

[33] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. A Practical Experiment to Give Dynamic Semantics to a DSL for Telephony Services Development. Technical report, Laboratoire d'Informatique de Nantes-Atlantique, 2006.

[34] T. Dinh-Trong, S. Ghosh, and R. France. JAL: Java like Action Language. Technical report, Department of Computer Science, Colorado State University, 2006.

[35] S. Efftinge and M. Völter. oAW xText: a Framework for Textual DSLs. In *Proceedings of the Modeling Symposium at Eclipse Summit*, 2006.

[36] H. Ehrig and C. Ermel. Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation. In *Proceedings of the 4th International Conference on Graph Transformation*, 2008. doi:10.1007/978-3-540-87405-8_14.

[37] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker. In *Proceedings of the 4th International Workshop on Rewriting Logic and Its Applications*, 2002. doi:10.1016/S1571-0661(05)82534-4.

[38] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull. Graphviz – Open Source Graph Drawing Tools. In *International Symposium on Graph Drawing*, 2002. doi:10.1007/3-540-45848-4_57.

[39] L.J.P. Engelen and M.G.J. van den Brand. Integrating Textual and Graphical Modelling Languages. In *Proceedings of the 9th Workshop on Language Descriptions, Tools, and Applications*, 2010. doi:10.1016/j.entcs.2010.08.035.

[40] L.J.P. Engelen and A.J. Wijs. Checking Property Preservation of Refining Transformations for Model-Driven Development. Technical report, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2012.

[41] S. Freeman and N. Pryce. Evolving an Embedded Domain-Specific Language in Java. In *Companion to the 21st ACM SIGPLAN symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006. doi:10.1145/1176617.1176735.

[42] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2011. doi:10.1007/978-3-642-19835-9_33.

[43] G. de Geest. Building a Framework to Support Domain-Specific Language Evolution. Master's thesis, Delft University of Technology, 2008.

[44] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification*, 2006.

[45] R.J. van Glabbeek. The Linear Time–Branching Time Spectrum II: The Semantics of Sequential Systems with Silent Moves. In *Proceedings of the 4th International Conference on Concurrency Theory*, 1993. doi:10.1007/3-540-57208-2_6.

[46] R.J. van Glabbeek, B. Luttik, and N. Trčka. Branching Bisimilarity with Explicit Divergence. *Fundamenta Informaticae*, 2009.

[47] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 1996. doi:10.1145/233551.233556.

[48] R. Gorrieri and A. Rensink. Action Refinement. In *Handbook of Process Algebra*. Elsevier, 2001.

[49] J.F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. van Weerdenburg, W. Wesselink, T. Willemse, and J. van der Wulp. The mCRL2 Toolset. In *Proceedings of the 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[50] A. Haase, M. Völter, S. Efftinge, and B. Kolb. Introduction to openArchitectureWare 4.1.2. In *Model-Driven Development Tool Implementers Forum (Co-Located with TOOLS 2007)*, 2007.

[51] S. Haustein and J. Pleumann. OCL as Expression Language in an Action Semantics Surface Language. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, 2004.

[52] M. Hennessy. *The Semantics of Programming Languages*. Wiley, 1990.

[53] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[54] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings of the 3rd International Spin Workshop*, 1997.

[55] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[56] J. Hooman and M.B. van der Zwaag. A Semantics of Communicating Reactive Objects with Timing. *International Journal on Software Tools for Technology Transfer*, 2006. doi:10.1007/s10009-005-0207-8.

[57] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In *Proceedings of the 8th International Conference on Integrated Formal Methods*, 2010. doi:10.1007/978-3-642-16265-7_14.

[58] S.C. Johnson. YACC: Yet Another Compiler-Compiler. Technical report, AT & T Bell Laboratories, 1975.

[59] A. Johnstone, P.D. Mosses, and E. Scott. An Agile Approach to Language Modelling and Development. *Innovations in Systems and Software Engineering*, 2010. doi:10.1007/s11334-009-0111-6.

[60] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. ATL: a QVT-like Transformation Language. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, 2006. doi:10.1145/1176617.1176691.

[61] F. Jouault, J. Bézivin, and I. Kurtev. TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, 2006. doi:10.1145/1173706.1173744.

[62] F. Jouault and I. Kurtev. Transforming Models with ATL. In *MoDELS 2005 Satellite Events*, 2005. doi:10.1007/11663430_14.

[63] M. Karaila. Evolution of a Domain Specific Language and its Engineering Environment – Lehman's Laws Revisited. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[64] G. Karsai, H. Krahn, C. Pinkernell, B. Rumpe, M. Schneider, and S. Völkel. Design Guidelines for Domain Specific Languages. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling*, 2009.

[65] G. Karsai and A. Narayanan. On the Correctness of Model Transformations in the Development of Embedded Systems. In *Proceedings of the 2006 Monterey Workshop*, 2007. doi:10.1007/978-3-540-77419-8_1.

[66] V. Kodaganallur. Incorporating Language Processing into Java Applications: a JavaCC Tutorial. *IEEE Software*, 2004. doi:10.1109/MS.2004.16.

[67] D.S. Kolovos, R.F. Paige, and F. Polack. The Epsilon Transformation Language. In *Proceedings of the 1st International Conference on Theory and Practice of Model Transformations*, 2008. doi:10.1007/978-3-540-69927-9_4.

[68] D. Kozen. Results on the Propositional $\mu$-calculus. *Theoretical Computer Science*, 1983. doi:10.1016/0304-3975(82)90125-6.

[69] I. Kurtev, K. van den Berg, and M. Akşit. UML to XML-Schema Transformation: a Case Study in Managing Alternative Model Transformations in MDA. In *Forum on Specification and Design Languages*, 2003.

[70] L. Lambers, H. Ehrig, and F. Orejas. Efficient Detection of Conflicts in Graph-Based Model Transformation. In *Proceedings of the 1st International Workshop on Graph and Model Transformation*, 2006. doi:10.1016/j.entcs.2006.01.017.

[71] F. Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-the-Fly Verification Methods. In *Proceedings of the 5th International Conference on Integrated Formal Methods*, 2005. doi:10.1007/11589976_6.

[72] C.F.J. Lange. Improving the Quality of UML Models in Practice. In *Proceedings of the 28th International Conference on Software Engineering*, 2006. doi:10.1145/1134285.1134472.

[73] C.F.J. Lange and M.R.V. Chaudron. An Empirical Assessment of Completeness in UML Designs. In *Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering*, 2004. doi:10.1049/ic:20040404.

[74] C.F.J. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers, and H.M. Dortmans. An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs. In *Proceedings of the 2nd Workshop on Consistency Problems in UML-based Software Development*, 2003.

[75] M.M. Lehman, J.F. Ramil, P. Wernick, D.E. Perry, and W.M. Turski. Metrics and Laws of Software Evolution – The Nineties view. In *Proceedings of the 4th IEEE International Software Metrics Symposium*, 1997. doi:10.1109/METRIC.1997.637156.

[76] H. Liang and J. Dingel. A Practical Evaluation of Using TXL for Model Transformation. In *Proceedings of the 1st International Conference on Software Language Engineering*, 2008. doi:10.1007/978-3-642-00434-6_16.

[77] J. Luoma, S. Kelly, and J.-P. Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. In *Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling*, 2004.

[78] R. Mateescu and A.J. Wijs. Property-Dependent Reductions for the Modal Mu-Calculus. In *Proceedings of the 18th International Workshop on Model Checking Software*, 2011. doi:10.1007/978-3-642-22306-8_2.

[79] T. Mens and P. van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 2006. doi:10.1016/j.entcs.2005.10.021.

[80] M. Mernik, J. Heering, and A.M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 2005. doi:10.1145/1118890.1118892.

[81] R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.

[82] P.D. Mosses. Component-Based Semantics. In *Proceedings of the 8th International Workshop on Specification and Verification of Component-Based Systems*, 2009. doi:10.1145/1596486.1596489.

[83] A. Narayanan and G. Karsai. Towards Verifying Model Transformations. In *Proceedings of the 5th International Workshop on Graph Transformation and Visual Modeling Techniques*, 2008. doi:10.1016/j.entcs.2008.04.041.

[84] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, January 2011.

[85] Object Management Group. MOF 2 XMI Mapping, Version 2.4.1, August 2011.

[86] Object Management Group. Unified Modeling Language, Version 2.4.1, August 2011.

[87] Object Management Group. Object Constraint Language, Version 2.3.1, January 2012.

[88] R.F Paige, J.S Ostroff, and P.J Brooke. Principles for Modeling Language Design. *Information and Software Technology*, 2000. doi:10.1016/S0950-5849(00)00109-9.

[89] R.F. Paige and A. Radjenovic. Towards Model Transformation with TXL. In *Proceedings of the Workshop on Metamodelling for MDA*, 2003.

[90] T.J. Parr and R.W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software — Practice and Experience*, 1995. doi:10.1002/spe.4380250705.

[91] M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 1995. doi:10.1145/203241.203251.

[92] B. Ploeger. Analysis of ACS using mCRL2. Technical report, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2009.

[93] G.D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 2004. doi:10.1016/j.jlap.2004.05.001.

[94] D. Plump. *Processes, Terms and Cycles: Steps on the Road to Infinity*, chapter Checking Graph-Transformation Systems for Confluence. Springer, 2005. doi:10.1007/11601548_16.

[95] A. Pnueli. The Temporal Logic of Programs. In *Proceedings fo the 18th Annual Symposium on Foundations of Computer Science*, 1977. doi:10.1109/SFCS.1977.32.

[96] T. Reenskaug. Models - Views - Controllers. Technical report, Xerox Parc, 1979.

[97] J.E. Rivera, F. Durán, and A. Vallecillo. Formal Specification and Analysis of Domain Specific Models Using Maude. *Simulation*, 2009. doi:10.1177/0037549709341635.

[98] V. Rusu. Embedding Domain-Specific Modelling Languages in Maude Specifications. *SIGSOFT Software Engineering Notes*, 2011. doi:10.1145/1921532.1921557.

[99] D.A. Sadilek and G. Wachsmuth. Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages. In *Proceedings of the European Conference on Model Driven Architecture: Foundations and Applications*, 2008. doi:10.1007/978-3-540-69100-6_5.

[100] D. Saha. An Incremental Bisimulation Algorithm. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science*, 2007. doi:10.1007/978-3-540-77050-3_17.

[101] B. Schätz. Verification of Model Transformations. *Electronic Communications of the EASST*, 2010.

[102] M. Scheidgen. Textual Modelling Embedded into Graphical Modelling. In *Proceedings of the 4th European Conference on Model Driven Architecture*, 2008. doi:10.1007/978-3-540-69100-6_11.

[103] D.C. Schmidt. Model-Driven Engineering. *Computer*, 2006. doi:10.1109/MC.2006.58.

[104] O.V. Sokolsky and S.A. Smolka. Incremental Model Checking in the Modal Mu-Calculus. In *Proceedings of the 6th International Conference on Computer Aided Verification*, 1994. doi:10.1007/3-540-58179-0_67.

[105] J. Sprinkle and G. Karsai. A Domain-Specific Visual Language for Domain Model Evolution. *Journal of Visual Languages and Computing*, 2004. doi:10.1016/j.jvlc.2004.01.006.

[106] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[107] G.M. Swamy. *Incremental Methods for Formal Verification and Logic Synthesis*. PhD thesis, University of California, 1996.

[108] B.D. Theelen, O. Florescu, M.C.W. Geilen, J. Huang, P.H.A. van der Putten, and J.P.M. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *Proceedings of IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2007. doi:10.1109/MEMCOD.2007.371231.

[109] D. Varró. Automated Formal Verification of Visual Modeling Languages by Model Checking. *Journal of Software and System Modeling*, 2004. doi:10.1007/s10270-003-0050-x.

[110] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.

[111] M. Völter. openArchitectureWare: a Flexible Open Source Platform for Model-Driven Software Development. In *Proceedings of the Eclipse Technology eXchange Workshop at the ECOOP 2006 Conference*, 2006.

[112] B.-Y. Wang. $\mu$-Calculus Model Checking in Maude. In *Proceedings of the 5th International Workshop on Rewriting Logic and Its Applications*, 2005. doi:10.1016/j.entcs.2004.06.025.

[113] J.M. Wing and M. Vaziri-Farahani. Model Checking Software Systems: A Case Study. *SIGSOFT Software Engineering Notes*, 1995. doi:10.1145/222132.222148.

# Appendix A

## Software Tools

*This appendix describes the software tools, languages, and platforms used to implement the languages and model transformations discussed in this thesis. Below, only short descriptions providing an overview of the most important features are given. The ways in which these tools, languages, and platforms have been applied are discussed extensively throughout the rest of this thesis.*

## A.1 ASF+SDF and the Meta-Environment

The language ASF+SDF [30] is a combination of the two formalisms ASF [16] and SDF [110]. The Syntax Definition Formalism (SDF) is a formalism for the definition of the syntax of context-free languages. The Algebraic Specification Formalism (ASF) is a formalism for the definition of conditional rewrite rules. Given a syntax definition in SDF of a source and target language, ASF can be used to define a transformation from the source language to the target language. In ASF, conditional rewrite rules are specified using the concrete syntax of the input and output languages.

Context-free languages are closed under union and, as a result of this, the SDF definitions of two languages can be combined to form the definition of a new context-free language, without altering the existing definitions. However, because ambiguities may arise after combining syntax definitions, it might be necessary to add constructs for disambiguation to a definition that combines existing languages. Using ASF in combination with SDF to implement transformations guarantees syntax safety. A transformation is syntax safe if it only accepts input that adheres to the syntax definition of the input language and always produces output adhering to the definition of the output language.

The ASF+SDF Meta-Environment [20] is an integrated development environment (IDE) for ASF+SDF. It has a graphical user interface that offers syntax-highlighting for the specification of SDF and ASF definitions, and an interpreter and debugger for the execution and debugging of ASF specifications. It can be used to create a command-line tool that parses and rewrites input adhering to the syntax definition of the input language and outputs the result. These command-line tools employ memoization, which ensures

that the result of a rewrite rule applied to a given term is computed only once. Both the ASF+SDF Meta-Environment and the command-line tools it generates use Annotated Terms (ATerms) [21] to represent terms internally. Because ATerms offer maximal subterm sharing, the internal representation of terms uses as little space as possible.

```
1  sorts
2    BoolCon BoolExp
3  context-free syntax
4    "true" | "false" -> BoolCon
5    BoolCon -> BoolExp
6    BoolExp "xor" BoolExp -> BoolExp {right}
7    eval(BoolExp) -> BoolCon
8  variables
9    "$BoolCon"[0-9]* -> BoolCon
10   "$BoolExp"[0-9]* -> BoolExp
```

Listing A.1: Part of an SDF definition that defines simple Boolean expressions

Listing A.1 shows a part of an SDF definition that defines a syntax for simple Boolean expressions. In ASF+SDF, each term conforms to a sort. On line 2, two such sorts are introduced, whose syntax is defined in lines 4 to 7. Line 4 states that a term of sort *BoolCon* is of the form "true" or "false". A term of sort *BoolCon* is also a valid term of sort *BoolExp*, as defined in line 5. Furthermore, line 6 specifies that two terms of sort *BoolCon* joined by the right-associative operator "xor" form a term of sort *BoolExp*. The signature of an evaluation function for Boolean expressions is specified on line 7. On lines 9 and 10, variables that represent terms of the aforementioned sorts are introduced.

```
1  [rule0]
2    eval($BoolCon) = $BoolCon
3
4  [rule1]
5    eval($BoolCon xor $BoolCon) = false
6
7  [rule2]
8    $BoolCon1 != $BoolCon2
9    ====>
10   eval($BoolCon1 xor $BoolCon2) = true
11
12 [default-rule]
13   $BoolCon1 := eval($BoolExp1),
14   $BoolCon2 := eval($BoolExp2)
15   ====>
16   eval($BoolExp1 xor $BoolExp2) = eval($BoolCon1 xor $BoolCon2)
```

Listing A.2: ASF rule for the evaluation of simple Boolean expressions

Listing A.2 shows the (conditional) rewrite rules that define how the simple Boolean expressions of Listing A.1 are evaluated. The first part of an ASF rule is optional and consists of the conditions of the rule, which are separated from the rest of the rule by an arrow (====>). Next, the left-hand side and right-hand side of the rule follow, separated by an equality sign. If a rule has no conditions or all its conditions hold, its application to a term results in replacing the left-hand side by the right-hand side. The rules described in this thesis use two kinds of conditions: (in)equality conditions and matching conditions.

An equality condition consists of a right-hand side and a left-hand side, separated by two equal signs (==). The condition holds if both sides can be matched. The right-hand side and left-hand side of an inequality condition are separated by an exclamation mark and an equal sign (!=), and it holds if both sides cannot be matched. Similarly, a matching condition consists of a right-hand side and a left-hand side, separated by a colon and an equal sign (:=). Also this type of condition holds if both sides can be matched. In this case, however, if both sides can be matched, the variables occurring at the left-hand side are instantiated accordingly. In Listing A.2, the first two rules have no conditions. The inequality condition of the third rule is shown on line 8, and the matching conditions of the last rule are shown on lines 13 and 14. On lines 1, 4, 7, and 12, the identifiers of the rules are shown. The last rule is only applied if non of the other rules are applicable, which is indicated by the fact that its identifier starts with "default".

## A.2 openArchitectureWare

The openArchitectureWare platform offers a number of tools related to model transformation: Xpand is used for model-to-text transformations, Xtext [35] is used for text-to-model transformations, and Xtend is used for model-to-model transformations. Here, the term "model" refers to an instance of an explicit metamodel. Execution of model-to-text transformations implemented with Xpand and model-to-model transformations implemented with Xtend can be automated using scripts for the Modeling Workflow Engine for Eclipse. Xpand and Xtend are based on the same type system and expression language. The type system offers simple types, such as string, Boolean, and integer, collection types, such as list and set, and the possibility to import metamodels. The expression language offers a number of basic constructs that can be used to create expressions, such as literals, operators, quantifiers, and switch expressions.

Currently, the platform no longer exists on its own and has become a part of the Eclipse Modeling Project[1] instead. It is implemented as a number of Eclipse plug-ins and is based on the Eclipse Modeling Framework (EMF) [106].

### A.2.1 Xpand

Xpand is a template-based language that generates text files given an EMF model. An Xpand template takes a metaclass and a list of parameters as input and produces output by executing a list of statements. There are a number of different types of statements, including one that saves the output generated by its statements to a file and one that triggers the execution of another template.

### A.2.2 Xtext

Xtext is a tool that parses text and converts it to an equivalent model, given a grammar describing the syntax of the input. Xtext uses ANTLR [90] to generate a parser that parses the textual representations of models. An Xtext specification consists of rules that define both a metamodel and a mapping from concrete syntax to this metamodel. Given a grammar, Xtext also generates an editor that provides features such as syntax highlighting and code completion.

[1] http://www.eclipse.org/modeling/

### A.2.3 Xtend

Xtend is a functional language for model transformation. It adds extensions to the basic expression language, which take a number of parameters as input and return the result of an expression. Because the extensions are not side-effect free, Xtend is not a pure functional language. Transformations implemented in Xtend are unidirectional, which means that they can only be used to transform models in a single direction. In other words, a transformation for given source and target metamodels can transform models conforming to the source metamodel into models conforming to the target metamodel, but not the other way around. The language can be used for in-place transformations, which modify a given model, as well as transformations that produce new models. Xtend is an interpreted language. It is supported by an IDE offering syntax highlighting, debugging, and code completion.

## A.3 ATL Transformation Language

The ATL Transformation Language (ATL) [60], previously known as the ATLAS transformation language, is another EMF based language for model transformation. Similarly to Xtend, ATL is also a unidirectional transformations language that offers both in-place transformations and creation of new models. The language provides both declarative and imperative constructs for the definition of model transformations. In contrast to Xtend, ATL does not have a native syntax for expressions, but uses the Object Constraint Language (OCL) [87] instead. ATL is supported by an IDE that offers debugging and syntax highlighting. A virtual machine is used to execute transformations after translating them to byte code. The execution of ATL transformations can be automated using ant tasks, which are small scripts that make it possible, for example, to compose transformations with and without saving the intermediate models.

## A.4 Dot and Graphviz

Dot is a language for graph visualization that is part of the Graphviz toolset [38]. Given a description of a graph written in Dot, Graphviz can visualize this graph as an image in various output formats. Graphviz employs layout algorithms to achieve optimal placement of nodes and edges. A graph description in Dot is a list of nodes and edges from node to node, combined with attributes that specify how particular nodes and edges should be displayed. These attributes define, for example, the color, width, height, and the type of lines used to draw these nodes and edges.

Although Dot is not designed specifically for applications in MDSE, we use it extensively for the visualization of graphical diagrams representing models. Currently, EMF based alternatives for graphical modeling do not provide the functionality required to create such diagrams.

# Appendix B

## Operational Semantics of SLCO

*This appendix discusses the formal operational semantics of SLCO and is based on the work described in Chapter 5. However, successful termination and time are not taken into account. A more concise description of the semantics presented in this appendix is given in Section 6.3.1. We start with a description of the syntax of SLCO, followed by a description of the rules that define its operational semantics. Finally, we discuss the initialization of the evaluation functions that are used in these rules.*

## B.1 Syntax

In this section, we use a variant of EBNF to define the syntax of SLCO. Although the use of quotation marks may suggest otherwise, the following syntax does not qualify as a concrete syntax for the language, because it does not assign a unique parse tree to each fragment of the language. Instead, the quotation marks are used to distinguish EBNF symbols from symbols of SLCO. For each element $e$ of a syntactic category, zero or more occurrences of $e$ are denoted by $e^*$, one or more occurrences are denoted by $e^+$, and zero or one occurrence is denoted by $[e]$.

The syntax of models $m \in \mathit{Models}$, classes $\mathit{class} \in \mathit{Classes}$, objects $\mathit{obj} \in \mathit{Objects}$, channels $\mathit{chan} \in \mathit{Channels}$, and variables $\mathit{var} \in \mathit{Variables}$ is defined as follows.

$$
\begin{array}{lll}
m & ::= & \mathit{mn}\ \mathit{class}^*\ \mathit{obj}^*\ \mathit{chan}^* \\
\mathit{class} & ::= & \mathit{cn}\ \mathit{var}^*\ \mathit{pn}^*\ \mathit{sm}^* \\
\mathit{obj} & ::= & \mathit{on}\ \text{“ : ”}\ \mathit{cn} \\
\mathit{chan} & ::= & \mathit{chn}\ \text{“(”}\ \mathit{type}^*\ \text{“)”}\ \mathit{chtype}\ \text{“}\textbf{from}\text{”}\ \mathit{on}\ \text{“.”}\ \mathit{pn}\ \text{“}\textbf{to}\text{”}\ \mathit{on}'\ \text{“.”}\ \mathit{pn}' \\
 & | & \mathit{chn}\ \text{“(”}\ \mathit{type}^*\ \text{“)”}\ \mathit{chtype}\ \text{“}\textbf{between}\text{”}\ \mathit{on}\ \text{“.”}\ \mathit{pn}\ \text{“}\textbf{and}\text{”}\ \mathit{on}'\ \text{“.”}\ \mathit{pn}' \\
\mathit{var} & ::= & \mathit{type}\ \mathit{vn}\ [\text{“ = ”}\ \mathit{ce}] \\
\mathit{type} & ::= & \text{“}\textbf{Boolean}\text{”} \quad | \quad \text{“}\textbf{Integer}\text{”} \quad | \quad \text{“}\textbf{String}\text{”} \\
\mathit{chtype} & ::= & \text{“}\textbf{sync}\text{”} \quad | \quad \text{“}\textbf{async lossless}\text{”} \quad | \quad \text{“}\textbf{async lossy}\text{”},
\end{array}
$$

where the structure of model names $\mathit{mn} \in \mathit{MN}$, class names $\mathit{cn} \in \mathit{CN}$, object names $\mathit{on} \in \mathit{ON}$, channel names $\mathit{chn} \in \mathit{CHN}$, port names $\mathit{pn} \in \mathit{PN}$, variable names $\mathit{vn} \in \mathit{VN}$, and

constant expressions $ce \in CE$ is left unspecified. We use a standard syntax for these concepts.

The syntax of state machines $sm \in StateMachines$, transitions $trans \in Transitions$, signal sending statements $send \in Statements$, signal reception statements $rec \in Statements$, and assignment statements $assign \in Statements$ is defined as follows.

$$
\begin{array}{lll}
sm & ::= & smn\ var^*\ \text{“}\mathbf{initial}\text{”}\ sn\ sn^*\ trans^* \\
trans & ::= & tn\ \text{“}\mathbf{from}\text{”}\ sn\ \text{“}\mathbf{to}\text{”}\ sn'\ [send \mid rec \mid assign \mid e] \\
send & ::= & \text{“}\mathbf{send}\text{”}\ sgn\ \text{“(”}\ e^*\ \text{“)”}\ \text{“}\mathbf{to}\text{”}\ pn \\
rec & ::= & \text{“}\mathbf{receive}\text{”}\ sgn\ \text{“(”}\ vn^*\ \text{“|”}\ e\ \text{“)”}\ \text{“}\mathbf{from}\text{”}\ pn \\
assign & ::= & vn\ \text{“}:=\text{”}\ e,
\end{array}
$$

where the structure of state machine names $smn \in SMN$, state names $sn \in SN$, transition names $tn \in TN$, signal names $sgn \in SGN$, and expressions $e \in Expressions$ is left unspecified. Again, we use a standard syntax for these concepts. Because we do not consider time and successful termination, the delay statement and the notion of final states are left out of this syntax definition.

## B.2 Semantics

We use a variant of structural operational semantics [52, 93] that relies heavily on valuation functions to define the semantics of SLCO. These valuation functions enable a compositional definition of the semantics. The potential behavior of an SLCO model is defined in terms of the potential behavior of the objects that constitute this model. In turn, the potential behavior of the objects is defined in terms of the potential behavior of their classes, which is defined in terms of the potential behavior of the state machines that constitute the classes. Finally, the potential behavior of a state machine is defined in terms of the potential behavior of the transitions of the state machine.

### B.2.1 Transitions

The potential behavior of a transition is defined by the relation

$$\rightarrow_{TRANS} \subseteq (Transitions \times SN \times V_{VARS} \times V_{VARS}) \times TL \times (SN \times V_{VARS} \times V_{VARS}),$$

where each function $v_{vars}$ from the set of partial functions $V_{VARS} = VN \nrightarrow CE$ maps variable names to constant expressions, and the syntax of transition labels $l \in TL$ is defined as follows.

$$
\begin{array}{lll}
l & ::= & \epsilon \\
 & \mid & \text{“}\mathbf{send}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”}\ \text{“}\mathbf{to}\text{”}\ pn \\
 & \mid & \text{“}\mathbf{receive}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”}\ \text{“}\mathbf{from}\text{”}\ pn \\
 & \mid & \text{“}\mathbf{send}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\
 & \mid & \text{“}\mathbf{receive}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\
 & \mid & \text{“}\mathbf{lost}\text{”}\ sgn\ \text{“(”}\ ce^*\ \text{“)”} \\
 & \mid & sgn\ \text{“(”}\ ce^*\ \text{“)”} \\
 & \mid & vn\ \text{“}:=\text{”}\ ce
\end{array}
$$

The relation $\rightarrow_{TRANS}$ is the least relation satisfying the following rules.

$$\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn', sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\epsilon}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle \tag{T1}$$

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} \mathbf{true}}{\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ e, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{\epsilon}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle} \tag{T2}$$

$$\frac{\langle assign, v_{vars}, v'_{vars}\rangle \xRightarrow{l}_{ASSIGN} \langle v''_{vars}, v'''_{vars}\rangle}{\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ assign, sn, v_{vars}, v'_{vars}\rangle \xrightarrow{l}_{TRANS} \langle sn', v''_{vars}, v'''_{vars}\rangle} \tag{T3}$$

$$\frac{\langle e^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXPS} ce^*}{\begin{array}{c}\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ \mathbf{send}\ sign(e^*)\ \mathbf{to}\ pn, sn, v_{vars}, v'_{vars}\rangle \\ \xrightarrow{\mathbf{send}\ sgn(ce^*)\ \mathbf{to}\ pn}_{TRANS} \langle sn', v_{vars}, v'_{vars}\rangle\end{array}} \tag{T4}$$

$$\frac{\langle ce^*, vn^*, v_{vars}, v'_{vars}\rangle \Rightarrow_{SUB} \langle v''_{vars}, v'''_{vars}\rangle, \quad \langle e, v''_{vars}, v'''_{vars}\rangle \Rightarrow_{EXP} \mathbf{true}}{\begin{array}{c}\langle tn\ \mathbf{from}\ sn\ \mathbf{to}\ sn'\ \mathbf{receive}\ sgn(vn^* \mid e)\ \mathbf{from}\ pn, sn, v_{vars}, v'_{vars}\rangle \\ \xrightarrow{\mathbf{receive}\ sgn(ce^*)\ \mathbf{from}\ pn}_{TRANS} \langle sn', v''_{vars}, v'''_{vars}\rangle\end{array}} \tag{T5}$$

Rule (T1) defines that a transition specification $tn$ **from** $sn$ **to** $sn'$ leads to a transition from state $sn$ to state $sn'$, leaving the valuation functions $v_{vars}$ and $v'_{vars}$ unchanged.

Rule (T2) defines that such a transition is also possible given a transition specification $tn$ **from** $sn$ **to** $sn'$ $e$, provided that the expression $e$ evaluates to **true**. This rule refers to a relation $\Rightarrow_{EXP} \subseteq (Expressions \times V_{VARS} \times V_{VARS}) \times CE$, which defines how expressions $e \in Expressions$ evaluate to constant expressions $ce \in CE$, given two valuation functions $v_{vars} \in V_{VARS}$ and $v'_{vars} \in V_{VARS}$. We do not specify the syntax of expressions, as mentioned above, and leave their semantics unspecified as well. We use a standard semantics for the evaluation of expressions, where two valuation functions are used to distinguish between the local variables of state machines and the global variables of objects. This distinction is discussed in further detail below.

Rule (T3) defines that the execution of an assignment statement as part of a transition leads to an update of the valuation functions $v_{vars}$ and $v'_{vars}$. The rule refers to the relation $\Rightarrow_{ASSIGN} \subseteq (Expressions \times V_{VARS} \times V_{VARS}) \times TL \times (V_{VARS} \times V_{VARS})$, which defines the details of this update. The relation $\Rightarrow_{ASSIGN}$ is the least relation satisfying the following rules.

$$\frac{\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} ce, \quad vn \in \mathrm{dom}(v'_{vars}), \quad v''_{vars} = v'_{vars}[ce/vn]}{\langle vn\ := \ e, v_{vars}, v'_{vars}\rangle \xRightarrow{vn := ce}_{ASSIGN} \langle v_{vars}, v''_{vars}\rangle}$$

$$\frac{\begin{array}{c}\langle e, v_{vars}, v'_{vars}\rangle \Rightarrow_{EXP} ce, \\ vn \notin \mathrm{dom}(v'_{vars}), \quad vn \in \mathrm{dom}(v_{vars}), \quad v''_{vars} = v_{vars}[ce/vn]\end{array}}{\langle vn\ := \ e, v_{vars}, v'_{vars}\rangle \xRightarrow{vn := ce}_{ASSIGN} \langle v''_{vars}, v'_{vars}\rangle}$$

In these rules, the distinction between local and global variables becomes apparent. In both rules, valuation function $v_{vars}$ maps the global variables to their values, and function $v'_{vars}$ maps the local variables to their values. If a local variable named $vn$ exists, denoted by $vn \in \mathrm{dom}(v'_{vars})$, then the valuation function $v'_{vars}$ is updated by mapping $vn$ to $ce$. Otherwise, if a global variable named $vn$ exists, the valuation function $v_{vars}$ is updated. We use $f[v/x]$ to denote the updated function $f$, where $f[v/x](x) = v$ and $f[v/x](y) = f(y)$ for all $y \neq x$.

Rule (T4) defines the semantics of transitions with statements that send signals. It refers to the relation $\Rightarrow_{EXPS} \subseteq (SEQ(Expressions) \times V_{VARS} \times V_{VARS}) \times SEQ(CE)$, which defines how a sequence of expressions is evaluated to a sequence of constant expressions,

given two valuation functions. The relation $\Rightarrow_{EXPS}$ is the least relation that satisfies the following rules.

$$\langle \epsilon, v_{vars}, v'_{vars} \rangle \Rightarrow_{EXPS} \epsilon$$

$$\frac{\langle e, v_{vars}, v'_{vars} \rangle \Rightarrow_{EXP} ce, \quad \langle e^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{EXPS} ce^*}{\langle e\ e^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{EXPS} ce\ ce^*}$$

Finally, an instance of rule (T5) exists for all $ce^* \in SEQ(CE)$. These instances define the semantics of transitions with signal reception statements. It refers to the relation $\Rightarrow_{SUB} \subseteq (CE \times VN \times V_{VARS} \times V_{VARS}) \times (V_{VARS} \times V_{VARS})$, which defines sequential updates of the values of a set of variables. The relation $\Rightarrow_{SUB}$ is the least relation satisfying the following rules.

$$\langle \epsilon, \epsilon, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v_{vars}, v'_{vars} \rangle$$

$$\frac{vn \in \text{dom}(v'_{vars}), \quad v''_{vars} = v'_{vars}[ce/vn], \quad \langle ce^*, vn^*, v_{vars}, v''_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}{\langle ce\ ce^*, vn\ vn^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}$$

$$\frac{\begin{array}{c} vn \notin \text{dom}(v'_{vars}), \quad vn \in \text{dom}(v_{vars}), \quad v''_{vars} = v_{vars}[ce/vn], \\ \langle ce^*, vn^*, v''_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle \end{array}}{\langle ce\ ce^*, vn\ vn^*, v_{vars}, v'_{vars} \rangle \Rightarrow_{SUB} \langle v'''_{vars}, v''''_{vars} \rangle}$$

Each instance of rule (T5) specifies that a statement **receive** $sgn(vn^* \mid e)$ **from** $pn$ is only enabled if expression $e$ evaluates to **true** after updating the values of the variables $vn^*$ to the constant expressions $ce^*$. This sequence of constant expressions represents the possible values of the arguments of signals sent by other objects.

## B.3 State Machines

The potential behavior of a state machine is defined by the relation

$$\rightarrow_{SM} \subseteq (StateMachines \times S_{SMS} \times V_{VARS} \times V_{SMS}) \times TL \times (S_{SMS} \times V_{VARS} \times V_{SMS}),$$

where each function $s_{sms}$ from the set of partial functions $S_{SMS} = SMN \nrightarrow SN$ maps state machine names to state names, and each function $v_{sms}$ from the set of partial functions $V_{SMS} = SMN \nrightarrow (VN \nrightarrow CE)$ maps state machine names to functions that map variable names to constant expression. The fact that state machine $smn$ is in state $sn$ is encoded as $s_{sms}(smn) = sn$ using a function $s_{sms} \in S_{SMS}$. Furthermore, the fact that variable $vn$ of state machine $smn$ has the value $ce$ is encoded as $v_{sms}(smn)(vn) = ce$ using a function $v_{sms} \in V_{SMS}$. The relation $\rightarrow_{SM}$ is the least relation satisfying the following rule.

$$\frac{\begin{array}{c} trans \in trans^*, \\ \langle trans, s_{sms}(smn), v_{vars}, v_{sms}(smn) \rangle \xrightarrow{l}_{TRANS} \langle sn, v'_{vars}, v''_{vars} \rangle, \\ s'_{sms} = s_{sms}[sn/smn], \quad v'_{sms} = v_{sms}[v''_{vars}/smn] \end{array}}{\langle smn\ var^*\ states\ trans^*, s_{sms}, v_{vars}, v_{sms} \rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms} \rangle} \quad \text{(SM)}$$

We use $e \in e^*$ to denote $\exists e', e''\ .\ e^* \equiv e'\ e\ e''$, for each element $e$ of a syntactic category.

Rule (SM) defines that if one of the transitions of a state machine can go from state $s_{sms}(smn)$ to state $sn$ while performing an action represented by $l$, then this state machine can make a transition to the same state from state $s_{sms}(smn)$ while performing the same action.

### B.3.1 Classes

The potential behavior of a class is defined by the relation

$$\rightarrow_{CLASS} \subseteq (Classes \times S_{SMS} \times V_{VARS} \times V_{SMS}) \times TL \times (S_{SMS} \times V_{VARS} \times V_{SMS}).$$

The relation $\rightarrow_{CLASS}$ is the least relation satisfying the following rule.

$$\frac{sm \in sm^*, \quad \langle sm, s_{sms}, v_{vars}, v_{sms}\rangle \xrightarrow{l}_{SM} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle}{\langle cn\ var^*\ port^*\ sm^*, s_{sms}, v_{vars}, v_{sms}\rangle \xrightarrow{l}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle} \tag{C}$$

Rule (C) defines that the potential behavior of a class is derived from the potential behavior of the state machines of that class.

### B.3.2 Objects

The potential behavior of a set of objects is defined by the relation

$$\begin{aligned}\rightarrow_{OBJS} \subseteq\ & (Objects \times Classes \times Channels \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B) \\ & \times TL \times (S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B),\end{aligned}$$

where each function $v_{glob}$ from the set of partial functions $V_{GLOB} = ON \nrightarrow (VN \nrightarrow CE)$ maps object names to functions that map variable names to constant expressions, each function $v_{loc}$ from the set of partial functions $V_{LOC} = ON \nrightarrow (SMN \nrightarrow (VN \nrightarrow CE))$ maps object names to functions that map state machine names to functions that map variable names to constant expressions, each function $s_{objs}$ from the set of partial functions $S_{OBJS} = ON \nrightarrow (SMN \nrightarrow SN)$ maps object names to functions that map state machine names to state names, and each function $b$ from the set of partial functions $B = (CHN \times ON \times ON) \nrightarrow (SGN \times SEQ(CE)) \cup \{\mathbf{nil}\}$ maps tuples consisting of a channel name and two object names to the constant nil or a tuple consisting of a signal name and a sequence of constant expressions. The functions in $V_{GLOB}$ encode valuations of global variables, and the functions in $V_{LOC}$ encode valuations of local variables. The set of functions $S_{OBJS}$ is an extension of the set $S_{SMS}$. The fact that state machine *smn* of object *on* is in state *sn* is encoded as $s_{objs}(on)(smn) = sn$ using a function $s_{objs} \in S_{OBJS}$. The functions in $B$ are used to encode the content of a set of buffers. The fact that the buffer corresponding to the channel *chn* that connects objects $on_1$ and $on_2$ is empty is encoded as $b(chn, on_1, on_2) = \mathbf{nil}$ using a function $b \in B$. In the remainder of this section, we discuss a number of the rules that define this relation.

The following rule defines the part of the semantics of sequences of objects related to assignment statements.

$$\frac{\begin{array}{c} on : cn \in obj^*, \quad cn\ var^*\ pn^*\ sm^* \in class^*, \\ \langle cn\ var^*\ pn^*\ sm^*, s_{objs}(on), v_{glob}(on), v_{loc}(on)\rangle \\ \xrightarrow{vn := ce}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on], \quad v'_{glob} = v_{glob}[v_{vars}/on], \quad v'_{loc} = v_{loc}[v_{sms}/on] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{vn := ce}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O1}$$

Rule (O1) specifies that a sequence of objects can perform an assignment $vn := ce$ if one of the objects in the sequence is an instance of a class that can perform this assignment.

The following rules defines the part of the semantics of sequences of objects related to synchronous communication.

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O2}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync between}\ on_1.pn_1\ \textbf{and}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O3}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad on_2 : cn_2 \in obj^*, \\ cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{sync between}\ on_2.pn_2\ \textbf{and}\ on_1.pn_1 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s'_{sms}, v'_{vars}, v'_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1][s'_{sms}/on_2], \\ v'_{glob} = v_{glob}[v'_{vars}/on_2], \quad v'_{loc} = v_{loc}[v'_{sms}/on_2] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{sn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b\rangle} \tag{O4}$$

Rule (O2) specifies synchronous communication between two objects over a unidirectional channel, and Rules (O3) and (O4) specify synchronous communication over bidirectional channels. Because sending a signal does not affect the valuation of variables, the updates of $v_{glob}$ and $v_{loc}$ do not take object $on_1$ into account.

The following rules define asynchronous communication over a lossless channel.

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossless from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1], \\ b(\langle chn, on_1, on_2\rangle) = \textbf{nil}, \quad b' = b[\langle sgn, ce^*\rangle/\langle chn, on_1, on_2\rangle] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{\textbf{send}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v_{glob}, v_{loc}, b'\rangle} \quad \text{(O5)}$$

$$\frac{\begin{array}{c} on_2 : cn_2 \in obj^*, \quad cn_2\ var_2^*\ pn_2^*\ sm_2^* \in class^*, \\ chn(type^*)\ \textbf{async lossless from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_2\ var_2^*\ pn_2^*\ sm_2^*, s_{objs}(on_2), v_{glob}(on_2), v_{loc}(on_2)\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)\ \textbf{from}\ pn_2}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_2], \\ v'_{glob} = v_{glob}[v_{vars}/on_2], \quad v'_{loc} = v_{loc}[v_{sms}/on_2], \\ b(\langle chn, on_1, on_2\rangle) = \langle sgn, ce^*\rangle, \quad b' = b[\textbf{nil}/\langle chn, on_1, on_2\rangle] \end{array}}{\begin{array}{c} \langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \\ \xrightarrow{\textbf{receive}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle \end{array}} \quad \text{(O6)}$$

We only give the rules related to unidirectional channels. The rules for bidirectional channels are similar, however, and can be derived from the rules given above.

Rule (O5) specifies that a signal is placed in the buffer that corresponds to an asynchronous channel if an object sends this signal over the channel. This is only possible if the buffer is empty when the statement is executed. Rule (O6) specifies that a signal is removed from the buffer that corresponds to an asynchronous channel if an object that is connected to this channel is able to receive this signal.

The following rules define how asynchronous communication over a lossy channel differs from asynchronous communication over a lossless channel.

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossy from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, B\rangle \xrightarrow{\textbf{lost}\ sgn(ce^*)}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, B\rangle} \quad \text{(O7)}$$

$$\frac{\begin{array}{c} on_1 : cn_1 \in obj^*, \quad cn_1\ var_1^*\ pn_1^*\ sm_1^* \in class^*, \\ chn(type^*)\ \textbf{async lossy from}\ on_1.pn_1\ \textbf{to}\ on_2.pn_2 \in chan^*, \\ \langle cn_1\ var_1^*\ pn_1^*\ sm_1^*, s_{objs}(on_1), v_{glob}(on_1), v_{loc}(on_1)\rangle \\ \xrightarrow{\textbf{send}\ sgn(ce^*)\ \textbf{to}\ pn_1}_{CLASS} \langle s_{sms}, v_{vars}, v_{sms}\rangle, \\ s'_{objs} = s_{objs}[s_{sms}/on_1], \\ b(\langle chn, on_1, on_2\rangle) = \langle sgn', ce^{*\prime}\rangle, \quad b' = b[\langle sgn, ce^*\rangle/\langle chn, on_1, on_2\rangle] \end{array}}{\langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{\textbf{lost}\ sgn'(ce^{*\prime})}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle} \quad \text{(O8)}$$

Besides the rules shown above, additional rules exist that complete the definition of asynchronous communication over lossy channels. These rules are similar to the rules

defining communication over asynchronous, lossless channels and can be derived from those rules.

Rule (O7) specifies that a signal sent over a lossy channel may get lost. In this rule, the function representing the buffer is unchanged. Rule (O8) specifies an alternative way of losing signals. It shows that a signal sent over a lossy channel can be placed in the corresponding buffer, even if this buffer already contains a signal. The new signal replaces the existing signal, which means that the original signal is lost.

### B.3.3 Models

Finally, the potential behavior of a model is defined by the relation

$$\begin{aligned}\rightarrow_{MODEL} \subseteq\ & (\mathit{Models} \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B) \\ & \times \mathit{TL} \times (\mathit{Models} \times S_{OBJS} \times V_{GLOB} \times V_{LOC} \times B),\end{aligned}$$

which is the least relation satisfying the following rule.

$$\frac{\begin{array}{c} m \equiv mn\ obj^*\ class^*\ chan^*, \\ \langle obj^*, class^*, chan^*, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{OBJS} \langle s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle \end{array}}{\langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle \xrightarrow{l}_{MODEL} \langle m, s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle} \quad \text{(M)}$$

## B.4 Initialization

By specifying which configurations $\langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle$ and $\langle m, s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle$ are related, Rule (M) defines the steps that can be taken according to model $m$. The initial configuration is defined by choosing appropriate functions $s_{objs}$, $v_{glob}$, $v_{loc}$, and $b$. Below, we define a number of functions that map SLCO models to the functions that define the initial configurations of these models.

### B.4.1 Initial States

The function $S^M_{OBJS} : \mathit{Models} \rightarrow (ON \nrightarrow (SMN \nrightarrow SN))$ is defined as follows.

$$\begin{aligned}& S^M_{OBJS}(mn\ class^*\ obj^*\ chan^*) = \\ & \quad \{(on, S^C_{SMS}(cn\ var^*\ pn^*\ sm^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}\end{aligned}$$

It maps models to functions from $S_{OBJS}$, such that each name of a state machine is mapped to its initial state as defined by the model. The definition refers to the function $S^C_{SMS} : \mathit{Classes} \rightarrow (SMN \nrightarrow SN)$, which is defined as follows.

$$S^C_{SMS}(cn\ var^*\ pn^*\ sm^*) = \{(smn, sn) \mid smn\ \mathbf{initial}\ sn\ sn^* \in sm^*\}$$

This function maps classes to functions from $S_{SMS}$, such that each name of a state machine is mapped to its initial state as defined by the class.

### B.4.2 Initial Values of Variables

The function $V^M_{GLOB} : \mathit{Models} \rightarrow (ON \nrightarrow (VN \nrightarrow CE))$ is defined as follows.

$$\begin{aligned}& V^M_{GLOB}(mn\ class^*\ obj^*\ chan^*) = \\ & \quad \{(on, V^V(var^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}\end{aligned}$$

It maps models to functions from $V_{GLOB}$, such that each global variable is mapped to its initial value as specified by the model. The definition refers to a function $V^V$ : $SEQ(Variables) \rightarrow (VN \nrightarrow CE)$, which is defined as follows.

$$
\begin{aligned}
&V^V(var^*) = \\
&\quad \{(vn, \mathbf{false}) \mid \mathbf{Boolean}\ vn \in var^*\} \ \cup\ \{(vn, bc) \mid \mathbf{Boolean}\ vn = bc \in var^*\} \cup \\
&\quad \{(vn, 0) \mid \mathbf{Integer}\ vn \in var^*\} \ \cup\ \{(vn, ic) \mid \mathbf{Integer}\ vn = ic \in var^*\} \cup \\
&\quad \{(vn, \text{``''}) \mid \mathbf{String}\ vn \in var^*\} \ \cup\ \{(vn, sc) \mid \mathbf{String}\ vn = sc \in var^*\}
\end{aligned}
$$

This function maps sequences of variable declarations to functions that map variable names to the appropriate initial values as specified by the sequence of declarations.

Similar functions $V^M_{LOC} : Models \rightarrow (ON \nrightarrow (SMN \nrightarrow (VN \nrightarrow CE)))$ and $V^C$ : $Classes \rightarrow (SMN \nrightarrow (VN \nrightarrow CE))$ exist that are related to the initial values of local variables. These functions are defined as follows.

$$
\begin{aligned}
&V^M_{LOC}(mn\ class^*\ obj^*\ chan^*) = \\
&\quad \{(on, V^C(cn\ var^*\ pn^*\ sm^*)) \mid cn\ var^*\ pn^*\ sm^* \in class^* \wedge on : cn \in obj^*\}
\end{aligned}
$$

$$
V^C(cn\ var^*\ pn^*\ sm^*) = \{(smn, V^V(var^*) \mid smn\ var^*\ states\ trans^* \in sm^*\}
$$

### B.4.3 Buffers

The function $B^M : Models \rightarrow ((CHN \times ON \times ON) \nrightarrow (SGN \times SEQ(CE)) \cup \{\mathbf{nil}\})$ is defined as follows.

$$
\begin{aligned}
&B^M(mn\ class^*\ obj^*\ chan^*) = \{(\langle chn, on_1, on_2\rangle, \mathbf{nil}) \mid \\
&\quad chn(type^*)\ \mathbf{async\ lossless\ from}\ on_1.pn_1\ \mathbf{to}\ on_2.pn_2 \in chan^* \vee \\
&\quad chn(type^*)\ \mathbf{async\ lossless\ between}\ on_1.pn_1\ \mathbf{and}\ on_2.pn_2 \in chan^* \vee \\
&\quad chn(type^*)\ \mathbf{async\ lossless\ between}\ on_2.pn_2\ \mathbf{and}\ on_1.pn_1 \in chan^* \vee \\
&\quad chn(type^*)\ \mathbf{async\ lossy\ from}\ on_1.pn_1\ \mathbf{to}\ on_2.pn_2 \in chan^* \vee \\
&\quad chn(type^*)\ \mathbf{async\ lossy\ between}\ on_1.pn_1\ \mathbf{and}\ on_2.pn_2 \in chan^* \vee \\
&\quad chn(type^*)\ \mathbf{async\ lossy\ between}\ on_2.pn_2\ \mathbf{and}\ on_1.pn_1 \in chan^* \\
&\}
\end{aligned}
$$

It maps models to functions from $B$, such that each buffer corresponding to a channel in the model is empty.

# Appendix C

## Two Transformations for SLCO

*This appendix provides a detailed description of the two versions of the transformation that simulates synchronous communication over an asynchronous channel. An informal description of both versions of this transformation is given in Section 3.5.1.1. Additionally, Section 6.2 provides a concise discussion of the description presented in this appendix. We use the syntax defined in Appendix B to describe the two versions of the transformation.*

## C.1 Simple Transformation

The transformation $T^S_{as}$ takes a model and a channel name as input. If the channel is used for synchronous communication, it is replaced by an asynchronous channel, and the classes of the objects that are connected by the channel are adapted. It is defined as

$$\begin{array}{l} T^S_{as}(mn\ class^*\ obj^*\ chan^*, chn) = \\ \quad mn\ T^S_{CS}(class^*, cn_1, pn_1, cn_2, pn_2)\ obj^* \\ \quad chan^*_1\ chn(type^*)\ \textbf{async lossless between}\ on_1.\,pn_1\ \textbf{and}\ on_2.\,pn_2\ chan^*_2, \end{array}$$

if $chan^* \equiv chan^*_1\ chn(type^*)\ \textbf{sync from}\ on_1.\,pn_1\ \textbf{to}\ on_2.\,pn_2\ chan^*_2$, $obj^* \equiv obj^*_1\ on_1 : cn_1\ obj^*_2$, and $obj^* \equiv obj^*_3\ on_2 : cn_2\ obj^*_4$, and it is defined as

$$T^S_{as}(mn\ class^*\ obj^*\ chan^*, chn) = mn\ class^*\ obj^*\ chan^*$$

otherwise.

The transformation $T^S_{CS}$ defined below takes a sequence of classes, two class names, and two port names as input. It returns a sequence of classes. The classes with the names provided as input are modified, and all other classes remain intact. There are two cases for this transformation. In one case, the class with name $cn_1$ is first in the sequence of classes. In the other case, the class with name $cn_2$ is first.

$$\begin{array}{l} T^S_{CS}(class^*, cn_1, pn_1, cn_2, pn_2) = \\ \quad class^*_1\ T^S_C(class_1, pn_1)\ class^*_2\ T^S_C(class_2, pn_2)\ class^*_3, \end{array}$$

if $class^* \equiv class_1^*\ class_1\ class_2^*\ class_2\ class_3^*$, $class_1 \equiv cn_1\ pn_1^*\ var_1^*\ sm_1^*$, and $class_2 \equiv cn_2\ pn_2^*\ var_2^*\ sm_2^*$, and

$$\begin{aligned}&T^S_{CS}(class^*, cn_1, pn_1, cn_2, pn_2) =\\ &\quad class_1^*\ T^S_C(class_2, pn_2)\ class_2^*\ T^S_C(class_1, pn_1)\ class_3^*,\end{aligned}$$

if $class^* \equiv class_1^*\ class_2\ class_2^*\ class_1\ class_3^*$, $class_1 \equiv cn_1\ pn_1^*\ var_1^*\ sm_1^*$, and $class_2 \equiv cn_2\ pn_2^*\ var_2^*\ sm_2^*$.

The transformation $T^S_C$ defined below takes a class and a port name as input and returns a class with modified state machines. Only the state machines that send or receive signals over the given port are modified.

$$T^S_C(cn\ pn^*\ var^*\ sm^*, pn) = cn\ pn^*\ var^*\ T^S_{SMS}(sm^*, pn)$$

The transformation $T^S_{SMS}$ defined below takes a sequence of state machines and a port name as input, and returns a modified sequence of state machines. Only the state machines that send or receive signals over the given port are modified. There are two cases for this transformation. One case deals with empty sequences of state machines and one case with non-empty sequences of state machines.

$$\begin{aligned}T^S_{SMS}(\epsilon, pn) &= \epsilon\\ T^S_{SMS}(sm\ sm^*, pn) &= T^S_{SM}(sm, pn)\ T^S_{SMS}(sm^*, pn)\end{aligned}$$

The transformation $T^S_{SM}$ defined below takes a state machine and a port name as input, and returns a state machine with modified transitions and a number of additional states. Only the transitions that send or receive signals over the given port are modified. There are two cases for this transformation. One case deals with state machines without final states, and the other with state machines with final states.

$$T^S_{SM}(smn\ var^*\ \textbf{initial}\ sn\ sn^*\ trans^*, pn) = smn\ var^*\ \textbf{initial}\ sn\ sn^*\ sn_1^*\ trans_1^*,$$

where $\langle sn_1^*, trans_1^*\rangle = T^S_{TS}(trans^*, pn)$, and

$$\begin{aligned}&T^S_{SM}(smn\ var^*\ \textbf{initial}\ sn\ sn^*\ \textbf{final}\ sn^+\ trans^*, pn) =\\ &\quad smn\ var^*\ \textbf{initial}\ sn\ sn^*\ sn_1^*\ \textbf{final}\ sn^+\ trans_1^*,\end{aligned}$$

where $\langle sn_1^*, trans_1^*\rangle = T^S_{TS}(trans^*, pn)$. The syntax definition given in Appendix B is extended as follows to accommodate final states.

$$sm ::= smn\ var^*\ \text{“}\textbf{initial}\text{”}\ sn\ sn^*\ [\text{“}\textbf{final}\text{”}\ sn^+]\ trans^*$$

The transformation $T^S_{TS}$ defined below takes a sequence of transitions and a port name as input, and returns a modified sequence of transitions and a set of new states. Only the transitions that send or receive signals over the given port are modified. There are two cases for this transformation. One case deals with empty sequences of transitions and the other with non-empty sequences.

$$\begin{aligned}T^S_{TS}(\epsilon, pn) &= \langle\epsilon, \epsilon\rangle\\ T^S_{TS}(trans\ trans^*, pn) &= \langle sn_1^*\ sn_2^*, trans_1^*\ trans_2^*\rangle,\end{aligned}$$

where $\langle sn_1^*, trans_1^*\rangle = T^S_T(trans, pn)$ and $\langle sn_2^*, trans_2^*\rangle = T^S_{TS}(trans^*, pn)$.

The transformation $T_T^S$ defined below takes a transition and a port name as input, and returns a set of states and a set of transitions. Only transitions that have a signal reception trigger or a send signal statement are replaced by new transitions. All other transitions are left unaltered. There are three cases for this transformation. One case deals with transitions that send a signal, one case deals with transitions that receive signals, and one case deals with all other transitions.

$$\begin{array}{l} T_T^S(tn \textbf{ from } sn_1 \textbf{ to } sn_2 \textbf{ send } sgn() \textbf{ to } pn, pn) = \langle \\ \quad sn_3, \\ \quad tn_1 \textbf{ from } sn_1 \textbf{ to } sn_3 \textbf{ send } sgn_1() \textbf{ to } pn \\ \quad tn_2 \textbf{ from } sn_3 \textbf{ to } sn_2 \textbf{ receive } sgn_2() \textbf{ from } pn \\ \rangle, \end{array}$$

where $sn_3$ is a fresh state name, $tn_1$ and $tn_2$ are fresh transition names, $sgn_1 \equiv$ "$s_$"$+sgn$, and $sgn_2 \equiv$ "$a_$" $+ sgn$,

$$\begin{array}{l} T_T^S(tn \textbf{ from } sn_1 \textbf{ to } sn_2 \textbf{ receive } sgn() \textbf{ from } pn, pn) = \langle \\ \quad sn_3, \\ \quad tn_1 \textbf{ from } sn_1 \textbf{ to } sn_3 \textbf{ receive } sgn_1() \textbf{ from } pn \\ \quad tn_2 \textbf{ from } sn_3 \textbf{ to } sn_2 \textbf{ send } sgn_2() \textbf{ to } pn \\ \rangle, \end{array}$$

where $sn_3$ is a fresh state name, $tn_1$ and $tn_2$ are fresh transition names, $sgn_1 \equiv$ "$s_$"$+sgn$, and $sgn_2 \equiv$ "$a_$" $+ sgn$, and

$$T_T^S(trans, pn) = \langle \epsilon, trans \rangle,$$

for all other transitions.

## C.2 General Transformation

The transformation $T_{as}^G$ takes a model and a channel name as input and returns a modified model. If the channel name refers to a synchronous channel, it is replaced by an asynchronous channel in the resulting model, and the classes of the objects that communicate over this channel are modified accordingly. It is defined as

$$\begin{array}{l} T_{as}^G(mn\ class^*\ obj^*\ chan^*, chn) = \\ \quad mn\ T_{CS}^G(class^*, cn_1, pn_1, cn_2, pn_2)\ obj^* \\ \quad chan_1^*\ chn(type^*) \textbf{ async lossless between } on_1.\, pn_1 \textbf{ and } on_2.\, pn_2\ chan_2^*, \end{array}$$

if $chan^* \equiv chan_1^*\ chn(type^*)$ **sync from** $on_1.\, pn_1$ **to** $on_2.\, pn_2\ chan_2^*$, $obj^* \equiv obj_1^*\ on_1 : cn_1\ obj_2^*$, and $obj^* \equiv obj_3^*\ on_2 : cn_2\ obj_4^*$, and it is defined as

$$T_{as}^G(mn\ class^*\ obj^*\ chan^*, chn) = mn\ class^*\ obj^*\ chan^*$$

otherwise.

The transformation $T_{CS}^G$ defined below takes a sequence of classes, two class names $cn_1$ and $cn_2$, and two port names as input and returns a sequence of modified classes. Only the classes named $cn_1$ and $cn_2$ are modified. This transformation has two cases. One case

is concerned with sequences of classes in which the sending class is first in the sequence, and the other case is concerned with sequences of classes in which the other class is first.

$$T^{G}_{CS}(class^*, cn_1, pn_1, cn_2, pn_2) = \\ class^*_1 \; T^{SEND}_{C}(class_1, pn_1) \; class^*_2 \; T^{REC}_{C}(class_2, pn_2) \; class^*_3,$$

if $class^* \equiv class^*_1 \; class_1 \; class^*_2 \; class_2 \; class^*_3$, $class_1 \equiv cn_1 \; pn^*_1 \; var^*_1 \; sm^*_1$, and $class_2 \equiv cn_2 \; pn^*_2 \; var^*_2 \; sm^*_2$, and

$$T^{G}_{CS}(class^*, cn_1, pn_1, cn_2, pn_2) = \\ class^*_1 \; T^{REC}_{C}(class_2, pn_2) \; class^*_2 \; T^{SEND}_{C}(class_1, pn_1) \; class^*_3,$$

if $class^* \equiv class^*_1 \; class_2 \; class^*_2 \; class_1 \; class^*_3$, $class_1 \equiv cn_1 \; pn^*_1 \; var^*_1 \; sm^*_1$, and $class_2 \equiv cn_2 \; pn^*_2 \; var^*_2 \; sm^*_2$.

The transformation $T^{SEND}_{C}$ defined below takes a class and a port name $pn$ as input and returns a modified class. The modified class has an extra integer variable and an extra state machine. The state machines that send signals via port $pn$ are modified, and the others are left intact.

$$T^{SEND}_{C}(cn \; pn^* \; var^* \; sm^*, pn) = \\ cn \; pn^* \; var^* \; \textbf{Integer} \; vn = 0 \; T^{SEND}_{SMS}(sm^*, pn, vn) \; T^{READER}_{SM}(pn, vn, sgn^*),$$

where $vn$ is a fresh variable name and $sgn^* \equiv T^{NAMES}_{SMS}(sm^*)$.

The transformation $T^{SEND}_{SMS}$ defined below takes a sequence of state machines, a port name $pn$, and a variable name as input and modifies the state machines that send signals over port $pn$.

$$T^{SEND}_{SMS}(\epsilon, pn, vn) = \epsilon$$

$$T^{SEND}_{SMS}(sm \; sm^*, pn, vn) = T^{SEND}_{SM}(sm, pn, vn) \; T^{SEND}_{SMS}(sm^*, pn, vn)$$

The transformation $T^{SEND}_{SM}$ defined below takes a state machine, a port name, and a variable name as input and returns a modified state machine. This transformation has two cases. One case is concerned with state machines with final states, and the other case is concerned with state machines without final states.

$$T^{SEND}_{SM}(smn \; var^* \; \textbf{initial} \; sn \; sn^* \; \textbf{final} \; sn^+ \; trans^*, pn, vn) = \\ smn \; var^* \; \textbf{initial} \; sn \; sn^* \; sn^*_1 \; \textbf{final} \; sn^+ \; trans^*_1,$$

where $\langle sn^*_1, trans^*_1 \rangle = T^{SEND}_{TS}(trans^*, pn, vn)$, and

$$T^{SEND}_{SM}(smn \; var^* \; \textbf{initial} \; sn \; sn^* \; trans^*, pn, vn) = \\ smn \; var^* \; \textbf{initial} \; sn \; sn^* \; sn^*_1 \; trans^*_1,$$

where $\langle sn^*_1, trans^*_1 \rangle = T^{SEND}_{TS}(trans^*, pn, vn)$.

The transformation $T^{SEND}_{TS}$ defined below takes a sequence of transitions, a port name, and a variable name as input and returns a set of states and transitions. The resulting transitions are used to replace the transitions provided as input.

$$T^{SEND}_{TS}(\epsilon, pn, vn) = \langle \epsilon, \epsilon \rangle$$

$$T^{SEND}_{TS}(trans \; trans^*, pn, vn) = \langle sn^*_1 \; sn^*_2, trans^*_1 \; trans^*_2 \rangle,$$

where $\langle sn^*_1, trans^*_1 \rangle = T^{SEND}_{T}(trans, pn, vn)$ and $\langle sn^*_2, trans^*_2 \rangle = T^{SEND}_{TS}(trans^*, pn, vn)$.

The transformation $T_T^{SEND}$ defined below takes a transition, a port name, and a variable name as input and returns a set of states and transitions.

$$
\begin{array}{l}
T_T^{SEND}(tn\ \mathbf{from}\ sn_1\ \mathbf{to}\ sn_2\ \mathbf{send}\ sgn()\ \mathbf{to}\ pn, pn, vn) = \langle \\
\quad sn_3\ sn_4\ sn_5\ sn_6\ sn_7, \\
\quad tn_1\ \mathbf{from}\ sn_1\ \mathbf{to}\ sn_3\ vn == 0 \\
\quad tn_2\ \mathbf{from}\ sn_3\ \mathbf{to}\ sn_4\ \mathbf{send}\ sgn(1)\ \mathbf{to}\ pn \\
\quad tn_3\ \mathbf{from}\ sn_4\ \mathbf{to}\ sn_5\ vn == 2 \\
\quad tn_4\ \mathbf{from}\ sn_5\ \mathbf{to}\ sn_6\ \mathbf{send}\ sgn(3)\ \mathbf{to}\ pn \\
\quad tn_5\ \mathbf{from}\ sn_6\ \mathbf{to}\ sn_2\ vn == 0 \\
\quad tn_6\ \mathbf{from}\ sn_4\ \mathbf{to}\ sn_7\ vn := 2 \\
\quad tn_7\ \mathbf{from}\ sn_7\ \mathbf{to}\ sn_1\ \mathbf{send}\ sgn(4)\ \mathbf{to}\ pn \\
\rangle,
\end{array}
$$

where $sn_3$, $sn_4$, $sn_5$, $sn_6$, and $sn_7$ are fresh state names and $tn_1$, $tn_2$, $tn_3$, $tn_4$, $tn_5$, $tn_6$, and $tn_7$ are fresh transition names, and

$$T_T^{SEND}(trans, pn, vn) = \langle \epsilon, trans \rangle$$

otherwise.

Similar to the transformations defined above, transformations $T_C^{REC}$, $T_{SMS}^{REC}$, $T_{SM}^{REC}$, $T_{TS}^{REC}$, and $T_T^{REC}$ exists, which modify the classes, state machines, and transitions that receive signals from a given port. Transformation $T_T^{REC}$ is defined as follows.

$$
\begin{array}{l}
T_T^{REC}(tn\ \mathbf{from}\ sn_1\ \mathbf{to}\ sn_2\ \mathbf{receive}\ sgn()\ \mathbf{from}\ pn, pn, vn) = \langle \\
\quad sn_3\ sn_4\ sn_5\ sn_6\ sn_7, \\
\quad tn_1\ \mathbf{from}\ sn_1\ \mathbf{to}\ sn_3\ vn == 1 \\
\quad tn_2\ \mathbf{from}\ sn_3\ \mathbf{to}\ sn_4\ \mathbf{send}\ sgn(2)\ \mathbf{to}\ pn \\
\quad tn_3\ \mathbf{from}\ sn_4\ \mathbf{to}\ sn_5\ vn == 3 \\
\quad tn_4\ \mathbf{from}\ sn_5\ \mathbf{to}\ sn_2\ \mathbf{send}\ sgn(0)\ \mathbf{to}\ pn \\
\quad tn_5\ \mathbf{from}\ sn_4\ \mathbf{to}\ sn_1\ vn == 4 \\
\quad tn_6\ \mathbf{from}\ sn_1\ \mathbf{to}\ sn_6\ vn == 4 \\
\quad tn_7\ \mathbf{from}\ sn_6\ \mathbf{to}\ sn_7\ vn := 3 \\
\quad tn_8\ \mathbf{from}\ sn_7\ \mathbf{to}\ sn_1\ \mathbf{send}\ sgn(0)\ \mathbf{to}\ pn \\
\rangle,
\end{array}
$$

where $sn_3$, $sn_4$, $sn_5$, $sn_6$, and $sn_7$ are fresh state names and $tn_1$, $tn_2$, $tn_3$, $tn_4$, $tn_5$, $tn_6$, $tn_7$, and $tn_8$ are fresh transition names, and

$$T_T^{REC}(trans, pn, vn) = \langle \epsilon, trans \rangle$$

otherwise.

### C.2.1 Auxiliary State Machine

The transformations $T_C^{SEND}$ and $T_C^{REC}$ introduce an auxiliary state machine, which is generated by the transformation $T_{SM}^{READER}$ defined below. This transformation takes a port name, a variable name, and a sequence of signal names as input and returns a new state machine. This state machine has a transition for each of the signal names provided as input.

$$T_{SM}^{READER}(pn, vn, sgn^*) = smn\ \mathbf{initial}\ sn\ \ T_{TS}^{READER}(pn, vn, sgn^*, sn),$$

where *smn* is a fresh state machine name and *sn* is a fresh state name.

The transformation $T_{TS}^{READER}$ defined below takes a port name, a variable name, a state name, and a sequence of signal names as input and returns a list of transitions. There are two cases for this transformation. One case deals with empty sequences of transitions and one with non-empty sequences.

$$T_{TS}^{READER}(pn, vn, sgn^*, sn) = \epsilon$$
$$T_{TS}^{READER}(pn, vn, sgn\ sgn^*, sn) = T_{T}^{READER}(pn, vn, sgn, sn)\ T_{TS}^{READER}(pn, vn, sgn^*, sn)$$

The transformation $T_{T}^{READER}$ defined below takes a port name, a variable name, a signal name, and a state name as input and returns a transition. This transition has the state *sn* as its source and target state, and receives signals named *sgn* via port *pn*.

$$T_{T}^{READER}(pn, vn, sgn, sn) = tn\ \textbf{from}\ sn\ \textbf{to}\ sn\ \textbf{receive}\ sgn(vn)\ \textbf{from}\ pn,$$

where *tn* is a fresh transition name.

### C.2.2 Signal Names

The state machine generated by transformation $T_{SM}^{READER}$ contains a transition for each signal sent over the channel that is replaced by transformation $T_{as}^{G}$. The transformation $T_{SMS}^{NAMES}$ defined below takes a sequence of state machines and a port name *pn* as input, and returns the signal names that are sent and received via port *pn*.

$$T_{SMS}^{NAMES}(\epsilon, pn) = \epsilon$$
$$T_{SMS}^{NAMES}(sm\ sm^*, pn) = T_{SM}^{NAMES}(sm, pn, )\ T_{SMS}^{NAMES}(sm^*, pn)$$

The transformation $T_{SM}^{NAMES}$ defined below takes a state machine and a port name *pn* as input, and returns the signal names that are sent and received via port *pn*.

$$T_{SM}^{NAMES}(smn\ var^*\ states\ trans^*) = T_{TS}^{NAMES}(trans^*, pn)$$

The transformation $T_{TS}^{NAMES}$ defined below takes a sequence of transitions and a port name *pn* as input, and returns the signal names that are sent and received via port *pn*.

$$T_{TS}^{NAMES}(\epsilon, pn) = \epsilon$$
$$T_{TS}^{NAMES}(trans\ trans^*, pn) = T_{T}^{NAMES}(trans, pn)\ T_{TS}^{NAMES}(trans^*, pn)$$

The function $T_{T}^{NAMES}$ defined below takes a transition and a port name *pn* as input. If the transition sends or receive a signal via port *pn*, the signal name is returned.

$$T_{T}^{NAMES}(tn\ \textbf{from}\ sn_1\ \textbf{to}\ sn_2\ \textbf{send}\ sgn()\ \textbf{to}\ pn, pn) = sgn$$
$$T_{T}^{NAMES}(tn\ \textbf{from}\ sn_1\ \textbf{to}\ sn_2\ \textbf{receive}\ sgn()\ \textbf{from}\ pn, pn) = sgn$$

and

$$T_{T}^{NAMES}(trans, pn) = \epsilon$$

for all other transitions.

# Appendix D

## Correctness of a Transformation

*This appendix presents a correctness proof of the simple version of the transformation for SLCO that replaces a synchronous channel with an asynchronous channel. We use the operational semantics of SLCO presented in Appendix B and the formal description of the transformation presented in Section C.1. A more concise description of this proof is given in Section 6.3.2.*

## D.1 Correctness Proof

In this appendix, we reason about an SLCO model $m = mn\ class^*\ obj^*\ chan^*$ and a unidirectional, synchronous channel $chan = chn()$ **sync from** $on_1.pn_1$ **to** $on_2.pn_2 \in chan^*$. As discussed in Chapter 6, transformation $T^S_{as}$ is considered to be correct if $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar [47] for some appropriate label-renaming function $\rho$, where $LTS(m)$ refers to the labeled transition system (LTS) of model $m$ as defined by the operational semantics described in Appendix B. First, we discuss the label-renaming function $\rho$ that corresponds to transformation $T^S_{as}$. Second, we list the conditions for model $m$ that must hold to ensure that $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar. Finally, we show that a branching bisimulation between $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ exists by defining a relation $R$ and showing that $R$ is a branching bisimulation according to the definition given in Section 6.3.2.

### D.1.1 Label-Renaming Function

The definition of transformation $T^S_{as}$ given in Appendix C.1 defines that all transitions in model $m$ that send or receive signals over channel $chan$ are replaced by transitions that send or receive signals with a modified name. Let $Sgn \subseteq SGN$ be the set of all signal names that are passed over channel $chan$ in model $m$. Without loss of generality, we assume that all other channels in model $m$ are used to pass signals with names not in $Sgn$. If this is not the case, the signals passed over these other channels can be renamed. After applying transformation $T^S_{as}$ to model $m$, the set of names of signals that are passed

over the modified channel $chan'$ is $Sgn' \cup Sgn''$, where $Sgn' = \{\text{“}s_\text{”} + sgn \mid sgn \in Sgn\}$ and $Sgn'' = \{\text{“}a_\text{”} + sgn \mid sgn \in Sgn\}$. The label-renaming function $\rho$ that corresponds to transformation $T^S_{as}$ is defined as follows, for every $sgn' \equiv \text{“}s_\text{”}+sgn$ and $sgn'' \equiv \text{“}a_\text{”}+sgn$ such that $sgn \in Sgn$.

- $\rho(\textbf{send}\ sgn'()) = \tau$
- $\rho(\textbf{receive}\ sgn'()) = sgn()$
- $\rho(\textbf{send}\ sgn''()) = \tau$
- $\rho(\textbf{receive}\ sgn''()) = \tau$

Renaming $\rho$ is straightforwardly extended on LTSs. By renaming the labels **receive** $sgn'()$ to $sgn()$, for every $sgn' \equiv \text{“}s_\text{”} + sgn$ such that $sgn \in Sgn$, we indicate that these labels represent successful communication. The other labels are renamed to $\tau$ because they represent the implicit synchronization in the source model and should result in unobservable behavior of the target model.

## D.1.2 Applicability Conditions

To ensure that $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar, a number of conditions must hold for model $m$. As mentioned above, the synchronous channel $chan$ connects two objects $obj_1$ and $obj_2$. Each object $obj_i$ is named $on_i$ and is an instance of class $class_i$ with name $cn_i$, for $i = 1, 2$. We require that the following conditions hold for $m$.

1. Channel $chan$ is unidirectional, as mentioned above.
2. At most one state machine $sm_1$ of object $obj_1$ sends signals over channel $chan$.
3. At most one state machine $sm_2$ of object $obj_2$ receives signals over channel $chan$.
4. Object $obj_1$ is the only instance of class $class_1$.
5. Object $obj_2$ is the only instance of class $class_2$.

A model can be adapted with the model transformations described in Section 3.5.1 if these conditions are not met. If the first condition does not hold, transformation $T_{uni}$ can be applied to replace a bidirectional channel with two unidirectional channels. Furthermore, transformation $T_{ex}$ can be applied if the second or third condition is not met. It ensures that each pair of state machines communicates over a channel that is used by these two state machines only. Finally, the transformation that clones classes can be applied if the last two conditions are not met.

Since only the behavior of the instances of classes $class_1$ and $class_2$ are affected by transformation $T^S_{as}$ and objects $obj_1$ and $obj_2$ are the only instances of these classes, the behavior of all other objects is unchanged. Furthermore, because state machines $sm_1$ and $sm_2$ are the only state machines that communicate over channel $chan$, the behavior of all other state machines is also unchanged. Finally, because channel $chan$ is unidirectional, state machine $sm_1$ can only send signals over this channel, and state machine $sm_2$ can only receive signals over this channel.

### D.1.3 Bisimulation Relation

We use $cf = \langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle$ to represent a configuration of $LTS(m)$ and $cf' = \langle T^S_{as}(m), s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle$ to represent a configuration of $\rho(LTS(T^S_{as}(m, chn)))$. According to the definition of branching bisimilarity, $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar if their initial configurations are branching bisimilar. For each configuration $cf = \langle m, s_{objs}, v_{glob}, v_{loc}, b\rangle$, including the initial configuration, a counterpart $cf' = \langle T^S_{as}(m), s'_{objs}, v'_{glob}, v'_{loc}, b'\rangle$ exists such that $s_{objs} = s'_{objs}$, $v_{glob} = v'_{glob}$, $v_{loc} = v'_{loc}$, $b'(\langle chn, on_1, on_2\rangle) = \mathbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = \mathbf{nil}$, and $b = b'$ otherwise. Transformation $T^S_{as}$ does not modify the initial states and the variables of a model. It does, however, replace a unidirectional, synchronous channel with a bidirectional, asynchronous channel. Therefore, according to the definitions concerning initialization provided in Section B.4, the initial configuration $cf'_i$ of $\rho(LTS(T^S_{as}(m, chn)))$ is the counterpart of the initial configuration $cf_i$ of $LTS(m)$ as discussed above. Thus, if we prove that a branching bisimulation exists between each configuration $cf$ and its counterpart $cf'$, we prove that $LTS(m)$ and $\rho(LTS(T^S_{as}(m, chn)))$ are branching bisimilar.

For each $sgn \in Sgn$, transformation $T^S_{as}$ modifies state machine $sm_1$ by replacing each transition

$$trans^s_{sgn} = tn \text{ **from** } sn^s_1 \text{ **to** } sn^s_2 \text{ **send** } sgn() \text{ **to** } pn$$

with two transitions

$$tn_1 \textbf{ from } sn^s_1 \textbf{ to } sn^s_{sgn} \textbf{ send } sgn_1() \textbf{ to } pn \text{ and}$$
$$tn_2 \textbf{ from } sn^s_{sgn} \textbf{ to } sn^s_2 \textbf{ receive } sgn_2() \textbf{ from } pn,$$

where $tn_1$ and $tn_2$ are fresh transition names, $sn^s_{sgn}$ is a fresh state name, $sgn_1 \equiv$ "$s_$" + $sgn$, and $sgn_2 \equiv$ "$a_$" + $sgn$. Similarly, for each $sgn \in Sgn$, each transition

$$trans^r_{sgn} = tn \textbf{ from } sn^r_1 \textbf{ to } sn^r_2 \textbf{ receive } sgn() \textbf{ from } pn$$

of state machine $sm_2$ is replaced with two transitions

$$tn_1 \textbf{ from } sn^r_1 \textbf{ to } sn^r_{sgn} \textbf{ receive } sgn_1() \textbf{ from } pn \text{ and}$$
$$tn_2 \textbf{ from } sn^r_{sgn} \textbf{ to } sn^r_2 \textbf{ send } sgn_2() \textbf{ to } pn,$$

where $tn_1$ and $tn_2$ are fresh transition names, $sn^r_{sgn}$ is a fresh state name, $sgn_1 \equiv$ "$s_$" + $sgn$, and $sgn_2 \equiv$ "$a_$" + $sgn$.

We define a relation $R$ between configurations as follows: $(cf, cf') \in R$ if and only if $v_{glob} = v'_{glob}$, $v_{loc} = v'_{loc}$, and one of the following four conditions holds.

1. (a) $s_{objs} = s'_{objs}$,
   (b) $b'(\langle chn, on_1, on_2\rangle) = \mathbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = \mathbf{nil}$, and $b' = b$ otherwise;
2. (a) $s_{objs}(on_1)(smn_1) = sn^s_1$, $s'_{objs}(on_1)(smn_1) = sn^s_{sgn}$, $s_{objs} = s'_{objs}$ otherwise,
   (b) $b'(\langle chn, on_1, on_2\rangle) = (sgn_1, \varepsilon)$, $b'(\langle chn, on_2, on_1\rangle) = \mathbf{nil}$, and $b' = b$ otherwise,

   if there is a transition $trans^s_{sgn}$ from state $sn^s_1$ and $sgn_1 \equiv$ "$s_$" + $sgn$;
3. (a) $s_{objs}(on_1)(smn_1) = sn^s_2$, $s_{objs}(on_2)(smn_2) = sn^r_2$, $s'_{objs}(on_1)(smn_1) = sn^s_{sgn}$, $s'_{objs}(on_2)(smn_2) = sn^r_{sgn}$, $s_{objs} = s'_{objs}$ otherwise,
   (b) $b'(\langle chn, on_1, on_2\rangle) = \mathbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = \mathbf{nil}$, and $b' = b$ otherwise,

   if there is a transition $trans^s_{sgn}$ to state $sn^s_2$ and a transition $trans^r_{sgn}$ to state $sn^r_2$;

4. (a) $s_{objs}(on_1)(smn_1) = sn^s_2$, $s'_{objs}(on_1)(smn_1) = sn^s_{sgn}$, $s_{objs} = s'_{objs}$ otherwise,
   (b) $b'(\langle chn, on_1, on_2\rangle) = \mathbf{nil}$, $b'(\langle chn, on_2, on_1\rangle) = (sgn_2, \varepsilon)$, and $b' = b$ otherwise,

   if there is a transition $trans^s_{sgn}$ to state $sn^s_2$ and $sgn_2 \equiv$ "a_" + $sgn$.

We prove that $R$ is a branching bisimulation by distinguishing two cases. First, we consider the case in which $(cf_1, cf'_1) \in R$ and $cf_1 \xrightarrow{l}_{MODEL} cf_2$. We distinguish four cases.

1. We assume that $(cf_1, cf'_1) \in R$ according to the first condition stated above and distinguish between the cases $l \not\equiv sgn()$ and $l \equiv sgn()$, for any $sgn \in Sgn$.

   - In the first case, the transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ cannot be the result of synchronous communication between state machines $sm_1$ and $sm_2$. Thus, the transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ is the result of some of the transitions of these two state machines that are unaffected by the transformation or one or more transitions of some of the other state machines in the model, which are also unaffected by the transformation. Because transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ is the result of behavior that is unaffected by the transformation, a transition $cf'_1 \xrightarrow{l}_{MODEL} cf'_2$ can also be made from configuration $cf'_1$, and $(cf_2, cf'_2) \in R$ according to the first condition.
   - In the second case, the transition $cf_1 \xrightarrow{sgn()}_{MODEL} cf_2$ is the result of synchronous communication between state machines $sm_1$ and $sm_2$. According to the SOS rule for synchronous communication, state machine $sm_1$ has to be in a state $sn^s_1$ with an outgoing transition $trans^s_{sgn}$, and state machine $sm_2$ has to be in a state $sn^r_1$ with an outgoing transition $trans^r_{sgn}$. After transformation, according to the SOS rules for asynchronous communication, state machine $sm_1$ can make a transition from such a state $sn^s_1$ to a state $sn^s_{sgn}$ and send a signal over channel $chan'$. This results in a transition $cf'_1 \xrightarrow{\tau}_{MODEL} cf'_2$, where the label $\tau$ is the result of renaming the label **send** $sgn_1()$, and $sgn_1 \equiv$ "s_" + $sgn$. Furthermore, $(cf_1, cf'_2) \in R$ according to the second condition. Next, state machine $sm_2$ can make a transition to a state $sn^r_{sgn}$ and receive the signal sent by state machine $sm_1$. This results in a transition $cf'_2 \xrightarrow{sgn()}_{MODEL} cf'_3$, where the label $sgn()$ is the result of renaming the label **receive** $sgn_1()$, and $sgn_1 \equiv$ "s_"+$sgn$. In configuration $cf'_3$, state machine $sm_1$ is in state $sn^s_{sgn}$ and state machine $sm_2$ is in state $sn^r_{sgn}$. Thus, $(cf_2, cf'_3) \in R$ according to the third condition.

2. We assume that $(cf_1, cf'_1) \in R$ according to the second condition stated above and distinguish between the cases $l \not\equiv sgn()$ and $l \equiv sgn()$, for any $sgn \in Sgn$.

   - In the first case, the transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ cannot result from behavior of state machine $sm_1$, because this state machine can only send a signal over the synchronous channel $chan$ in a state $sn^s_1$ with an outgoing transition $trans^s_{sgn}$. Thus, the transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ must result from behavior of one or more other state machines. If the transition is the result of behavior of state machine $sm_2$, then it must be the result of one of the transitions of this state machine that are unaffected by the transformation, because the affected transitions can only lead to synchronous communication. Furthermore, all other state machines are unaffected by the transformation. Thus, the same behavior can be performed in the transformed model, leading to a transition $cf'_1 \xrightarrow{l}_{MODEL} cf'_2$. Furthermore, since the values of $s_{objs}(on_1)(smn_1)$ and $s'_{objs}(on_1)(smn_1)$ remain the same, $(cf_2, cf'_2) \in R$.

- In the second case, according to the SOS rule for synchronous communication, state machine $sm_2$ has to be in a state $sn_1^r$ with an outgoing transition $trans_{sgn}^r$. After transformation, according to the SOS rules for asynchronous communication, state machine $sm_2$ is able to receive the signal in the buffer that corresponds to channel $chan'$. This leads to a transition $cf_1' \xrightarrow{sgn()}_{MODEL} cf_2'$, where the label $sgn()$ is the result of renaming the label **receive** $sgn_1()$, and $sgn_1 \equiv$ "$s_$" $+ sgn$. In configuration $cf_2$, state machine $sm_1$ is in state $sn_2^s$ and state machine $sm_2$ is in state $sn_2^r$. Furthermore, in configuration $cf_2'$, state machine $sm_1$ is in state $sn_{sgn}^s$ and state machine $sm_2$ is in state $sn_{sgn}^r$. Thus, $(cf_2, cf_2') \in R$ according to the third condition.

3. We assume that $(cf_1, cf_1') \in R$ according to the third condition stated above. If transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ is the result of behavior of state machines that are unaffected by the transformation, then $cf_1' \xrightarrow{l}_{MODEL} cf_2'$ and $(cf_2, cf_2') \in R$. Otherwise, according to the SOS rules for asynchronous communication, state machine $sm_2$ can make a transition from a state $sn_{sgn}^r$ to a state $sn_2^r$ and send a signal over channel $chan'$, resulting in a transition $cf_1' \xrightarrow{\tau}_{MODEL} cf_2'$, where the label $\tau$ is the result of renaming the label **send** $sgn_2()$, and $sgn_2 \equiv$ "$a_$" $+ sgn$. Furthermore, $(cf_1, cf_2') \in R$ according to the fourth condition. Next, state machine $sm_1$ can make a transition from a state $sn_{sgn}^s$ to a state $sn_2^s$ and receive the signal sent over channel $chan'$, resulting in a transition $cf_2' \xrightarrow{\tau}_{MODEL} cf_3'$, where the label $\tau$ is the result of renaming the label **receive** $sgn_2()$, and $sgn_2 \equiv$ "$a_$" $+ sgn$. According to the first condition, $(cf_1, cf_3') \in R$. Following the reasoning for the first case, each transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ either corresponds to a transition $cf_3' \xrightarrow{l}_{MODEL} cf_4'$ with $(cf_2, cf_4') \in R$ or two transitions $cf_3' \xrightarrow{\tau}_{MODEL} cf_4'$ and $cf_4' \xrightarrow{l}_{MODEL} cf_5'$ with $(cf_1, cf_4') \in R$ and $(cf_2, cf_5') \in R$.

4. We assume that $(cf_1, cf_1') \in R$ according to the fourth condition stated above. If transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ is the result of behavior of state machine $sm_2$ or state machines that are unaffected by the transformation, then $cf_1' \xrightarrow{l}_{MODEL} cf_2'$ and $(cf_2, cf_2') \in R$. Otherwise, according to the SOS rules for asynchronous communication, state machine $sm_1$ can make a transition from a state $sn_{sgn}^s$ to a state $sn_2^s$ and receive the signal sent over channel $chan'$, resulting in a transition $cf_1' \xrightarrow{\tau}_{MODEL} cf_2'$, where the label $\tau$ is the result of renaming the label **receive** $sgn_2()$, and $sgn_2 \equiv$ "$a_$" $+ sgn$. According to the first condition, $(cf_1, cf_2') \in R$. Following a similar reasoning as in the case discussed above, transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ can be mimicked from configuration $cf_2'$.

Second, we consider the case in which $(cf_1, cf_1') \in R$ and $cf_1' \xrightarrow{l}_{MODEL} cf_2'$. We distinguish four cases.

1. We assume that $(cf_1, cf_1') \in R$ according to the first condition stated above. If $cf_1' \xrightarrow{\tau}_{MODEL} cf_2'$, then this transition must be the result of a transition of state machine $sm_1$ from a state $sn_1^s$ to a state $sn_{sgn}^s$, and the label $\tau$ must be the result of renaming the label **send** $sgn_1$, where $sgn_1 \equiv$ "$s_$"$+sgn$. In that case, $(cf_1, cf_2') \in R$ according to the second condition. Otherwise, the transition $cf_1' \xrightarrow{l}_{MODEL} cf_2'$ is the result of behavior of state machine $sm_2$ or one of the state machines that are unaffected by the transformation. In that case, the corresponding transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ must also exist for the original model, and $(cf_2, cf_2') \in R$ holds according to the first condition.

2. We assume that $(cf_1, cf'_1) \in R$ according to the second condition stated above. If $cf'_1 \xrightarrow{sgn()}_{MODEL} cf'_2$, then this transition must be the result of a transition of state machine $sm_2$ from a state $sn^r_1$ to a state $sn^r_{sgn}$, and the label $sgn()$ must be the result of renaming the label **receive** $sgn_1$, where $sgn_1 \equiv$ "$s_$" $+ sgn$. In that case, $cf'_1 \xrightarrow{sgn()}_{MODEL} cf'_2$ and $(cf_2, cf'_2) \in R$ according to the third condition. If this is not the case, then the transition $cf'_1 \xrightarrow{l}_{MODEL} cf'_2$ is the result of behavior of state machine $sm_2$ that is unaffected by the transformation or behavior of one of the state machines that are unaffected by the transformation. In that case, the corresponding transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ must also exist for the original model, and $(cf_2, cf'_2) \in R$ holds according to the second condition.

3. We assume that $(cf_1, cf'_1) \in R$ according to the third condition stated above. If $cf'_1 \xrightarrow{\tau}_{MODEL} cf'_2$, then this transition must be the result of a transition of state machine $sm_2$ from a state $sn^r_{sgn}$ to a state $sn^r_2$, and the label $\tau$ must be the result of renaming the label **send** $sgn_2$, where $sgn_2 \equiv$ "$a_$"$+sgn$. In that case, $(cf_1, cf'_2) \in R$ according to the fourth condition. Otherwise, the transition $cf'_1 \xrightarrow{l}_{MODEL} cf'_2$ is the result of behavior one of the state machines that are unaffected by the transformation. In that case, the corresponding transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ must also exist for the original model, and $(cf_2, cf'_2) \in R$ holds according to the third condition.

4. We assume that $(cf_1, cf'_1) \in R$ according to the fourth condition stated above. If $cf'_1 \xrightarrow{\tau}_{MODEL} cf'_2$, then this transition must be the result of a transition of state machine $sm_1$ from a state $sn^s_{sgn}$ to a state $sn^s_2$, and the label $\tau$ must be the result of renaming the label **receive** $sgn_2$, where $sgn_2 \equiv$ "$a_$"$+sgn$. In that case, $(cf_1, cf'_2) \in R$ according to the first condition. Otherwise, the transition $cf'_1 \xrightarrow{l}_{MODEL} cf'_2$ is the result of behavior of state machine $sm_2$ that is unaffected by the transformation or behavior of one of the state machines that are unaffected by the transformation. In that case, the corresponding transition $cf_1 \xrightarrow{l}_{MODEL} cf_2$ must also exist for the original model, and $(cf_2, cf'_2) \in R$ holds according to the fourth condition.

# Appendix E

## Case Studies Concerning Property Preservation

*This appendix presents the case studies of Section 8.5 in more detail. We describe the input models used for the experiments we performed and show how they are transformed.*

## E.1 ACS, 1394-fin, and Wafer Stepper

The ACS Manager along with Containers and Components is a part of the software of the ALMA project, carried out by the European Southern Observatory [92]. The intention of this project is to put more than 60 radio telescopes on a plane high up in the mountains of Chili for radio astronomy. A specification of this system is part of the official distribution of the mCRL2 toolset [49]. Figure E.1 describes a transformation of the receive action ($rcv$) into a more detailed procedure involving decompression of the received message. This rule was applied on the two components and one container present in the specification. The other parties in the two-way synchronizations were essentially left unchanged. However, by rewriting the action *send* to $send'$, we ensure that the rule system is terminating, confluent, and synchronization uniform. Furthermore, the rule system adds a synchronization rule stating that *decompress* can be fired by itself.

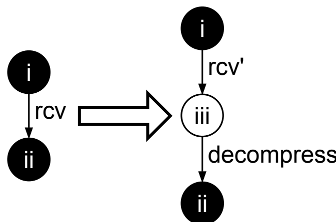



Figure E.1: A transformation rule refining the processing of received messages

The 1394-fin (Firewire) case and the Wafer Stepper case are two other mCRL2 specifications that have been transformed using rules very similar to this rule, but which involve different numbers of transitions.

## E.2 Broadcast

Broadcast is a system of fifteen processes communicating via three-party broadcast, i.e. three processes at a time synchronize simultaneously. Figure E.2 shows two pairs of three such processes. For each group of three processes, there is a synchronization rule that states that actions $a1$, $a2$, and $a3$ synchronize.

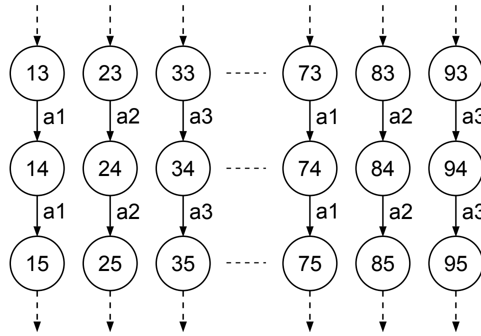



Figure E.2: Groups of three processes that communicate via broadcast

Due to restrictions imposed by an implementation platform, a transformation that breaks this down into a series of two-party synchronization might be desired. Three transformation rules that refine a model in this way are shown in Figure E.3. After transformation, new synchronization rules are introduced that define that $a1'$ and $a2'$, and $a2''$ and $a3'$ synchronize. This naive refinement does not preserve properties.

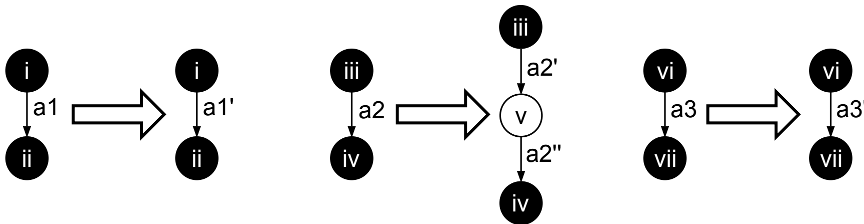



Figure E.3: Transformation rules that replace a three-party broadcast by pairwise communication

Improved versions of these transformation rules are shown in Figure E.4. Actions $a2$ and $a3$ are replaced by $a2'$ and $a3'$, respectively, to make the rule system terminating and confluent. After transformation, new synchronization rules are introduced that define that

- $m1a1$ and $m2a1$,
- $c1a1$ and $c2a1$,
- $a1a1$ and $a2a1$, and
- $a2'$ and $a3'$

synchronize. The dashed $\tau$-transitions in Figure E.4 indicate that this transformation is only property preserving if state $i$ is matched on states that are diverging.

To check whether the transformation rules of Figure E.4 preserve properties, a number of checks have to be performed. Figure E.5 show some of the LTSs that are used for

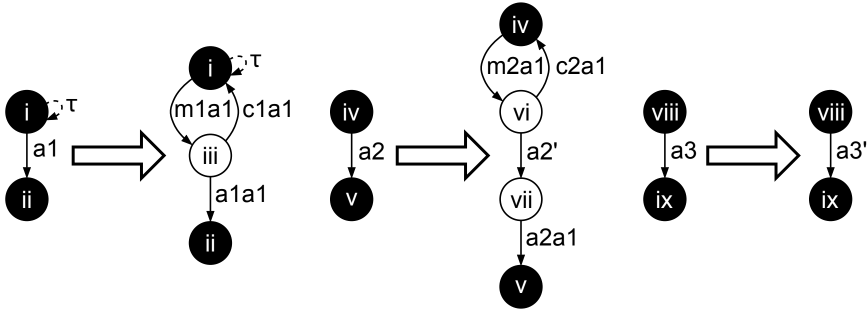



Figure E.4: Three improved transformation rules that replace a three-party broadcast by pairwise communication

these checks. The LTSs in Figure E.5 are created from the left-hand sides of the three transformation rules in Figure E.4. The tools Exp.Open and ltscompare of the mCRL2 toolkit cannot handle LTSs with multiple initial states. To be able to use these tools to perform the checks, one initial state is added to each of the LTSs, as well as $\tau$-transitions to the original initial states. Figure E.5 also show the $\kappa$-loops that are added to the original initial states. Each of the checks determines whether a network consisting of a combination of LTSs created from the left-hand sides of transformation rules is divergence-sensitive branching bisimilar with the network consisting of the corresponding LTSs created from the right-hand sides, after hiding the appropriate actions in both networks.

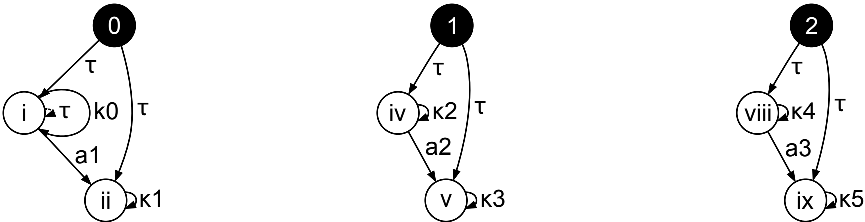



Figure E.5: Process LTSs of the left-hand sides of the transformation rules in Figure E.4

## E.3 Alternating Bit Protocol

Figure E.6 shows two components, $P$ and $Q$, which communicate via four buffers, *B1*, *B2*, *B3*, and *B4*. For the experiments described in Section 8.5, we analyzed a model containing five instances of such a communicating system operating concurrently.

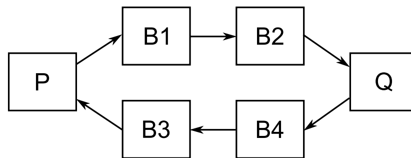
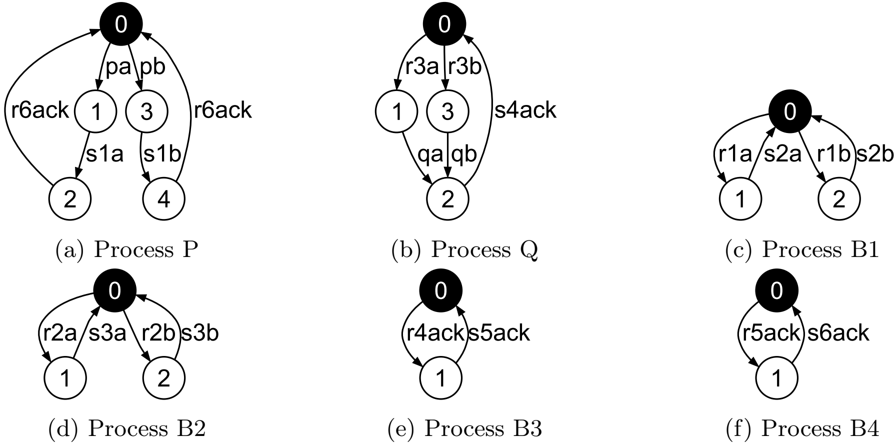



Figure E.6: Two components ($P$ and $Q$) that communicate via four buffers (*B1* to *B4*)

Figure E.7 show the process LTSs representing the six components. Process $P$ performs an action $pa$ or an action $qa$, and then communicates with component $Q$ via the buffers. After receiving either an $a$ or a $b$ from $P$, process $Q$ performs an action $qa$ or an action $qb$. Afterwards, $Q$ acknowledges the message reception. The numbers in the action labels represent the channels and indicate which actions synchronize. For example, actions $s1a$ and $r1a$ synchronize.




(a) Process P (b) Process Q (c) Process B1

(d) Process B2 (e) Process B3 (f) Process B4

Figure E.7: Process LTSs of the components shown in Figure E.6

Figure E.8 shows two transformation rules that replace two of the buffers by two processes that implement the alternating bit protocol. The alternating bit is encoded by the added suffixes "t" and "f" in the transition labels. To make the rule system synchronization uniform regarding the synchronization rules of the system, there are also transformation rules for the other processes that only perform some simple renaming, for instance to rename $pa$ to $pa'$. Lossy communication is simulated by adding two versions of the synchronization rules that specify how the actions representing communication between components *B1* and *B2* synchronize. One version specifies that a pair of such actions synchronize and lead to successful communication, and the other specifies that an action that represents sending a message can also be performed independently. The erroneous version of this transformation mentioned in Section 8.5 misses the loops on states $ix$ and $xiv$ that represent the reception of messages with the wrong bit.

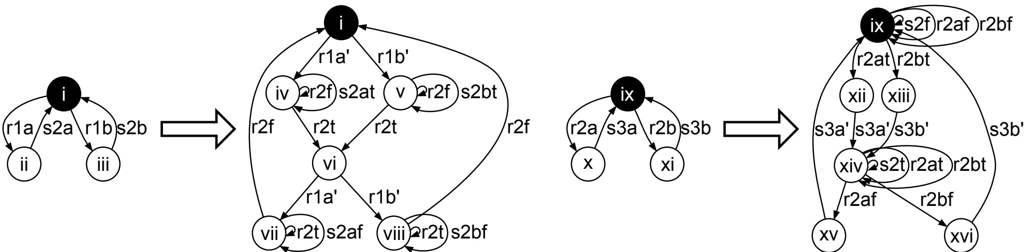



Figure E.8: Two transformation rules that replace buffers

# Summary


## From Napkin Sketches to Reliable Software

In the past few years, model-driven software engineering (MDSE) and domain-specific modeling languages (DSMLs) have received a lot of attention from both research and industry. The main goal of MDSE is generating software from models that describe systems on a high level of abstraction. DSMLs are languages specifically designed to create such models. High-level models are refined into models on lower levels of abstraction by means of model transformations.

The ability to model systems on a high level of abstraction using graphical diagrams partially explains the popularity of the informal modeling language UML. However, even designing simple software systems using such graphical diagrams can lead to large models that are cumbersome to create. To deal with this problem, we investigated the integration of textual languages into large, existing modeling languages by comparing two approaches and designed a DSML with a concrete syntax consisting of both graphical and textual elements. The DSML, called the Simple Language of Communicating Objects (SLCO), is aimed at modeling the structure and behavior of concurrent, communicating objects and is used as a case study throughout this thesis. During the design of this language, we also designed and implemented a number of transformations to various other modeling languages, leading to an iterative evolution of the DSML, which was influenced by the problem domain, the target platforms, model quality, and model transformation quality.

Traditionally, the state-space explosion problem in model checking is handled by applying abstractions and simplifications to the model that needs to be verified. As an alternative, we demonstrate a model-driven engineering approach that works the other way around using SLCO. Instead of making a concrete model more abstract, we refine abstract models by transformation to make them more concrete, aiming at the verification of models that are as close to the implementation as possible. The results show that it is possible to validate more concrete models when fine-grained transformations are applied instead of coarse-grained transformations.

Semantics are a crucial part of the definition of a language, and to verify the correctness of model transformations, the semantics of both the input and the output language must be formalized. For these reasons, we implemented an executable prototype of the semantics of SLCO that can be used to transform SLCO models to labeled transition systems (LTSs), allowing us to apply existing tools for visualization and verification of LTSs to SLCO models. For given input models, we can use the prototype in combination with these tools to show, for each transformation that refines SLCO models, that the input and output models exhibit the same observable behavior.

This, however, does not prove the correctness of these transformations in general. To prove this, we first formalized the semantics of SLCO in the form of structural operational semantics (SOS), based on the aforementioned prototype. Then, equivalence relations between LTSs were defined based on each transformation, and finally, these relations were shown to be either strong bisimulations or branching bisimulations. In addition to this approach, we studied property preservation of model transformations without restricting ourselves to a fixed set of transformations. Our technique takes a property and a transformation, and checks whether the transformation preserves the property. If a property holds for the initial model, which is often small and easy to analyze, and the property is preserved, then the refined model does not need to be analyzed too.

Combining the MDSE techniques discussed in this thesis enables generating reliable and correct software by means of refining model transformations from concise, formal models specified on a high level of abstraction using DSMLs.

# Curriculum Vitae

## Personal Information

*Name:* Lucas Johannes Petrus (Luc) Engelen

*Date of birth:* October 7, 1981

*Place of birth:* Nijmegen, the Netherlands

## Education

*Technische Informatica* *2000–2006*

Eindhoven University of Technology

Eindhoven, the Netherlands

*Atheneum* *1994–2000*

Jeanne D'Arc College

Maastricht, the Netherlands

## Professional Experience

*Researcher* *2012–present*

LaQuSo

Eindhoven, the Netherlands

*PhD candidate* *2008–2012*

Eindhoven University of Technology

Eindhoven, the Netherlands

*Junior researcher* *2006–2008*

Eindhoven University of Technology

Eindhoven, the Netherlands

# IPA Dissertation Series

## Titles in the IPA Dissertation Series since 2006

**E. Dolstra**. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

**R.J. Corin**. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

**P.R.A. Verbaan**. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

**K.L. Man and R.R.H. Schiffelers**. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

**M. Kyas**. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

**M. Hendriks**. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

**J. Ketema**. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

**C.-B. Breunesse**. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

**B. Markvoort**. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

**S.G.R. Nijssen**. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

**G. Russello**. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

**L. Cheung**. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

**B. Badban**. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

**A.J. Mooij**. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

**T. Krilavicius**. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

**M.E. Warnier**. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

**V. Sundramoorthy**. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

**B. Gebremichael**. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

**L.C.M. van Gool**. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

**C.J.F. Cremers**. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

**J.V. Guillen Scholten**. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

**H.A. de Jong**. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

**N.K. Kavaldjiev**. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

**M. van Veelen**. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

**T.D. Vu**. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

**L. Brandán Briones**. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

**I. Loeb**. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

**M.W.A. Streppel**. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

**N. Trčka**. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

**R. Brinkman**. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

**A. van Weelden**. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

**J.A.R. Noppen**. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

**R. Boumen**. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

**A.J. Wijs**. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

**C.F.J. Lange**. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

**T. van der Storm**. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

**B.S. Graaf**. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

**A.H.J. Mathijssen**. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

**D. Jarnikov**. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

**M.A. Abam**. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

**W. Pieters**. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

**A.L. de Groot**. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

**M. Bruntink**. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

**A.M. Marin**. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

**N.C.W.M. Braspenning**. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

**M. Bravenboer**. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

**M. Torabi Dashti**. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

**I.S.M. de Jong**. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

**I. Hasuo**. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

**L.G.W.A. Cleophas**. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

**I.S. Zapreev**. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

**M. Farshi**. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

**G. Gulesir**. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

**F.D. Garcia**. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

**P.E.A. Dürr**. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

**E.M. Bortnik**. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

**R.H. Mak**. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

**M. van der Horst**. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

**C.M. Gray**. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

**J.R. Calamé**. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

**E. Mumford**. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

**E.H. de Graaf**. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

**R. Brijder**. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

**A. Koprowski**. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

**U. Khadim**. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

**J. Markovski**. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

**H. Kastenberg**. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

**I.R. Buhan**. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

**R.S. Marin-Perianu**. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

**M.H.G. Verhoef**. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

**M. de Mol**. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

**M. Lormans**. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

**M.P.W.J. van Osch**. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

**H. Sozer**. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

**M.J. van Weerdenburg**. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

**H.H. Hansen**. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

**A. Mesbah**. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

**A.L. Rodriguez Yakushev**. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

**K.R. Olmos Joffré**. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

**J.A.G.M. van den Berg**. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

**M.G. Khatib**. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

**S.G.M. Cornelissen**. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

**D. Bolzoni**. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

**H.L. Jonker**. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

**M.R. Czenko**. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

**T. Chen**. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

**C. Kaliszyk**. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

**R.S.S. O'Connor**. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

**B. Ploeger**. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

**T. Han**. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

**R. Li**. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

**J.H.P. Kwisthout**. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

**T.K. Cocx**. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

**A.I. Baars**. *Embedded Compilers.* Faculty of Science, UU. 2009-25

**M.A.C. Dekker**. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

**J.F.J. Laros**. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

**C.J. Boogerd**. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

**M.R. Neuhäußer**. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

**J. Endrullis**. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

**T. Staijen**. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

**Y. Wang**. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

**J.K. Berendsen**. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

**A. Nugroho**. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

**A. Silva**. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

**J.S. de Bruin**. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

**D. Costa**. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

**M.M. Jaghoori**. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

**R. Bakhshi**. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

**B.J. Arnoldus**. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

**E. Zambon**. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

**L. Astefanoaei**. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

**J. Proença**. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

**A. Moralı**. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

**M. van der Bijl**. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

**C. Krause**. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

**M.E. Andrés**. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

**M. Atif**. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

**P.J.A. van Tilburg**. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

**Z. Protic**. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

**S. Georgievska**. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

**S. Malakuti**. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

**M. Raffelsieper**. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

**C.P. Tsirogiannis**. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

**Y.-J. Moon**. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

**R. Middelkoop**. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

**M.F. van Amstel**. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

**A.N. Tamalet**. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

**H.J.S. Basten**. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

**M. Izadi**. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

**L.C.L. Kats**. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

**S. Kemper**. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

**J. Wang**. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

**A. Khosravi**. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

**A. Middelkoop**. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

**Z. Hemel**. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

**T. Dimkov**. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

**S. Sedghi**. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

**F. Heidarian Dehkordi**. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

**K. Verbeek**. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

**D.E. Nadales Agut**. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

**H. Rahmani**. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

**S.D. Vermolen**. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

**L.J.P. Engelen**. *From Napkin Sketches to Reliable Software*. Faculty of Mathematics and Computer Science, TU/e. 2012-11