Bridging formal models : an engineering perspective

Citation for published version (APA):
Stappers, F. P. M. (2012). *Bridging formal models : an engineering perspective*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR738909

DOI:
10.6100/IR738909

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)

Bridging Formal Models
An Engineering Perspective

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 Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 8 november 2012 om 16.00 uur

door

Frank Petrus Maria Stappers

geboren te Weert

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr.ir. J.F. Groote
en
prof.dr. M.G.J. van den Brand

Copromotor:
dr.ir. M.A. Reniers

“If our lives don’t have meaning, what can we leave behind for those we care about?”
Michael C. Hall as *Dexter Morgan* (*Dexter*, 2011).

Promotor: prof.dr.ir. J.F. Groote (Technische Universiteit Eindhoven)
prof.dr. M.G.J. van den Brand (Technische Universiteit Eindhoven)
Copromotor: dr.ir. M.A. Reniers (Technische Universiteit Eindhoven)

Kerncommissie:
Prof.dr. J.J.M. Hooman (Radboud Universiteit Nijmegen)
Prof.dr. J.J. Lukkien (Technische Universiteit Eindhoven)
dr. W. Serwe (Centre de Recherche Inria, Grenoble, Rhône-Alpes)

IPA dissertation series 2012-12.

Typeset using LaTeX (TeXLive 2012).
Cover design by Rob Jacobs.
Printed by Printservice Eindhoven University of Technology.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics), supported by "*ITEA project TWINS: Optimizing Software Hardware Co-design Flow for Software Intensive Systems*" (No. 05004) and "*KWR project LithoSysSL: Language-oriented, Domain Specific Modeling Environments for the Specification Verification and Validation of Lithography Systems*" (No. 09124).

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-3263-6

Acknowledgments

It has been over five years ago that I started as a PhD-student. Jan Friso Groote asked me during one of the drinks at the Formal Methods Group, whether I would like to participate in the European ITEA2 Twins project. The start of this project and the ensuing five years finally led to this thesis. As science and experts constantly scan the horizon for new problems to be solved in both academia and industry, there are many people to whom I owe thanks.

First of all, I would like to express my gratitude towards my supervisors Jan Friso Groote and Michel Reniers. Jan Friso gave me the opportunity to conduct research and freely explore my fields of interest. Michel helped me to structure my chaotic ideas into human readable results. As co-author of almost all of the work in this thesis, it is more than safe to say that without Michel's help this thesis would have been a lot shorter. Furthermore, I thank them for the patience they had to listen to me. I enjoyed the countless discussions and numerous conversations we had.

I would also like to thank the reading committee for carefully reviewing my thesis: Mark van den Brand, Jozef Hooman, Johan Lukkien and Wendelin Serwe. They have provided me valuable comments and feedback that improved the manuscript substantially. I am also honored that Jean-Marie Jacquet agreed to serve as an opponent during the defense.

During my time as a PhD-student, I participated in the ITEA2 Twins project and the LithoSysSL project at ASML. Both projects allowed me to work with many inspiring people, led me to do some amazing assignments at different sites, and enabled me to directly discuss research and its practice with people that share similar interests. Here, I would like to thank all participants from the Twins Project for the wonderful times we had during the European meetings. My thanks go out to Suzana Andova, Jos Baeten, Istvan Nagy and Sven Weber for giving me the opportunity to participate in the LithoSysSL Project. The internship at ASML's Production Control Group repositioned my research direction. It showed me the kind of research that is important to industry. In particular, I am indebted to Sven as being my daily supervisor during the internship period, his enthusiasm for applying formal methods, and being a

committed co-author for the work presented in the second part of this thesis.

For my time at the university, I have spent the first two and a half years at the Design And Analysis Group, one year at the Formal Methods Group, and one and a half year at the Model Driven Software Engineering Group. Here, I had the opportunity to work with, and to meet fascinating people. I would like to say a word of thanks to all mCRL2 developers which I have met throughout the years, and all the people with which I had to share rooms: Jeroen van der Wulp, Aad Mathijssen, Doaa Hassan, Harsh Beohar, Tim Willemse and Maciej Gazda. Furthermore, I would like to thank all co-authors, people and companies that provided me content to work with: Suzana Andova, Sjoerd Cranen, Lou Dohmen, Jan Friso Groote, Jeroen Keiren, Aad Mathijssen, Maarten Meulen, Istvan Nagy, Bas Ploeger, Michel Reniers, Eugen Schindler, Klement Schindler, Lou Somers, Carst Tankink, Yaroslav Usenko, Sven Weber, Muck van Weerdenburg, Wieger Wesselink, Tim Willemse, Jeroen van der Wulp, NBG Industrial Automation and ASML. In particular, I would like to thank Jeroen Keiren and Sven Weber for spending their valuable time to read my thesis and provide helpful comments.

A special word of thanks I am allowed to all my family and friends for supporting me during my PhD candidacy.

Finally, I thank my parents, Albert and Ine, and my brother Mark for their unconditional love, care and support throughout the years.

Frank Stappers, Augustus 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Background

Modern systems are a synergistic combination of hardware components and governing control software. These systems find their applications in different fields ranging from consumer electronics to aerospace flight systems. Because these systems are increasingly facilitating our day-to-day services, they require to be functionally reliable.

By providing additional services the complexity of these systems grows. This means that it becomes harder to guarantee the absence of errors, as tests become more time-consuming and costly. Consequently, it becomes unfeasible to manually and exhaustively test all possible behavior. Hence, the chances increase that errors remain undetected in manufactured systems.

Since it is practically impossible, and in some cases even undesired, to test all behavior on a (physical) system, other techniques are required that ensure correctly operating systems. Here, *formal analysis* using *behavioral models* can assist. These models describe the essential behavior without ambiguity, which makes them suitable for different kinds of analysis, such as demonstrating use cases, running virtual simulations, or proving their correctness.

Within a field of a single engineering discipline many different formalisms, methods and tools are available to carry out such an analysis. The tools that support an analysis are as weak as the most difficult concept that needs to be specified and analyzed. Therefore, selecting an appropriate formalism that allows for the specification, the modeling, and the analysis of a system is a difficult task.

To assist engineers, this thesis puts forward two research questions, namely:

I *"What are practiced methods to create formal behavioral models, suitable for verification?"*, and

II *"Can we systematically administer formal techniques to translate behavioral models into a formalism that facilitates formal verification?"*

The first part of the thesis investigates the first research question. It sketches the difficulties that engineers face when hand-crafting formal models. With the help of a model engineering framework we formulate, position and explain the practical (bridging) problem statement when formal models are engineered for a behavioral analysis. The framework describes the different application environments and the model-to-model transformations (bridges) that connect them. Illustrated by several case studies, we show the usage, benefits and pitfalls for several of these bridges.

The second part of the thesis addresses the second research question. It investigates how the quality of formal models can be improved, while reducing the labor intensive task of hand-crafting formal models. It constitutes a technique that allows the engineering of formal models in a processable manner using the semantics of a language. The thesis explains the entire process. It starts with the formalization of an informal language. Then we present the technique that transforms a model, along with a formal language definition, into a formal model suitable for analysis. With the help of this technique, we finally analyze a formal language and its implementation. The language and implementation are used for the specification, modeling, validation and verification of system behavior.

1.2 Model Engineering Framework

In multidisciplinary system development, models are typically created in different languages to represent different views on a system. The way in which the different views are related is depicted in Figure 1.1, which presents the *model engineering framework*.

An *environment* defines a perspective where engineers specify, reason about, analyze, and possibly execute behavior that is associated with these models. An environment consists of at most three components. The components that are obligatory are a *language* component and a *model* component. The *engine* component is optional.

A *language* component is dissected into two parts, namely *syntax* and *semantics*. Syntax specifies the form of a language. Semantics specifies the meaning of a language. Both can be decomposed into two parts.

Syntax is decomposed into *abstract syntax* and *concrete syntax*.

- The abstract syntax represents the *mathematical notions* of a language, typically reflected by processable and algebraic tree-like data structures.
- The concrete syntax consists of the *visible notions* of a language including all the visible features of a language like parentheses and delimiters.

Semantics is decomposed into *static semantics* and *dynamic semantics*.

- The *static semantics* defines the restrictions on a structure of valid notions. Static semantics is used to e.g., check that identifiers occur in an appropriate context, or that variables are properly typed.

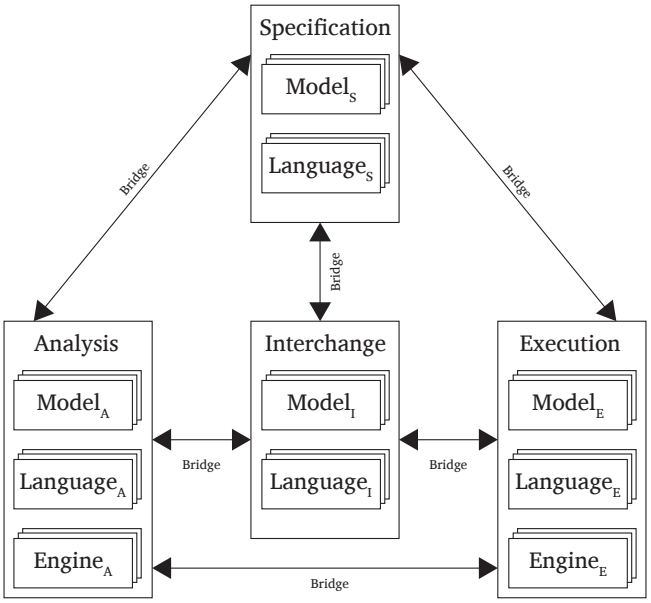


Figure 1.1 Bermuda Triangle of model engineering

- The *dynamic semantics* defines the computational model for the (composed) execution behavior of the language's abstract notions. The dynamic semantics can be defined in three different classes. *Operational semantics* is a commonly used class to express dynamic semantics. Examples of these notations are found in [Chu32, GKOT00, Plo04].

  Axiomatic semantics is an alternative class that assigns meaning to a set of abstract notions and their relations through predetermined assertions. An example is found in [Wil11].

  The final class is *denotational semantics*, which describes the effect of a set of abstract notions and their relations [Sco70] through another piece of syntax, possibly defined in another language. Examples of this notation are found in [Rep93, JRH+99, HVB00].

A *model* component is a behavioral description of a system prescribed by a grammar and is restricted to the static semantics of a language. A model is an instance of a language.

An (optional) *engine* component implements the (intended) dynamic semantics of a language. The semantic execution can be expressed in various ways, e.g., a computation plotted as a function over time, a list of operations performed by a system, or as transitions between states.

1.2.1 Environments

The model engineering framework consists of at most four environments. These environments are a specification environment, an analysis environment, an execution environment, and an optional interchange environment.

Specification Environment

The specification environment allows engineers to create their behavioral specifications (i.e., models) of a system. Depending on their purpose and the application, i.e., the domain, different specifications from different formalisms can be used. Typically, the models are geared towards a particular set of aspects for a domain. Hence, domain specific languages often facilitate model development.

In our view, the specification environment is solely used to specify models. Whenever we want to analyze or execute these models, they need to be first transformed to either the analysis or the execution environment, respectively.

Analysis Environment

The analysis environment allows engineers to analyze the dynamic behavior of a model. To determine if a model conforms to a set of properties, the model is either validated or verified. By *validation* one experimentally tests for a limited set of scenarios that a property holds. In practice, there are two forms, namely (i) simulation and (ii) testing [CGP99]. Simulation is a validation based on the execution of an analysis using (abstract) models. Testing is a validation based on the actual realization, e.g., the actual software or hardware modules. *Verification* provides a proof that a desired property holds for all possible conditions and for all possible scenarios w.r.t. a model. In practice, there are two forms, namely (i) model checking and (ii) theorem proving [CGP99, HR04]. Model checking consists of a systematic *exhaustive* (symbolic) exploration of the system's state space while dealing with a modal property. Formal proving relies on mathematical reasoning yielding a proof that a property holds.

Verification results only hold for the enclosed models. They do not guarantee a *correct implementation*. Because models often serve as guidelines for an implementation, the implementation itself needs to be validated separately.

Execution Environment

The execution environment allows engineers to execute models on a dedicated platform. Executable models are obtained when models are compiled into source or machine-specific code. A compilation enriches a model with the implementation details and the inner-workings of a platform's execution architecture. After the model has been compiled, it can be executed on the targeted platform.

Interchange Environment

The interchange environment provides engineers the (operational) re-use of models, offers inter-operability, or coordinates interaction between environments. An interchange environment is often described by a special language that incorporates concepts from different languages, results from agile development, or the attempts from earlier efforts to connect environments. Ideally, an interchange language is defined as a super-set language that spans over all of the languages that are used throughout a development process. Interchange environments often act as a pivot point for which transformations to and from other environments are created.

An example of a language that is used in the interchange environment is the Compositional Interchange Format (CIF) [BRSR07, BCN+09], which has a formal and compositional semantics based on (hybrid) transition systems and allows property-preserving model-to-model transformations between languages. Another language is the eXtensible Markup Language (XML) [BPSM+08], a markup language that defines a set of rules for encoding documents in a format that is both human- and machine-readable, originally designed for simplicity, generality, and usability over the internet.

1.2.2 Model Engineering Bridges

To share information between environments model-to-model transformations are required. A *model engineering bridge* denotes a model-to-model transformation between two (possibly) different environments. In this thesis we distinguish two kinds of bridges, namely *syntactic bridges* and *semantic bridges*.

A syntactic bridge transforms the (relevant) syntactic notions of one language into a (set of related) syntactic notions in another language without explicitly considering the semantics. A syntactic bridge produces *syntactically engineered models*.

A semantic bridge explicitly considers the semantics when mathematically transforming models. For the latter kind of bridges, we assume that the relationship between two languages is mathematically defined through the semantics, and could be automated based on its language definition. A semantic bridge produces *semantically engineered models*.

1.3 Problem Statement

To analyze a system, we need to create a model in a formalism that facilitates formal verification. Creating these bridges requires craftsmanship that is guided by the intuition of experts, or by engineers that require intensive education. Even though many systems have been formally verified, in both academic and industrial settings, and flaws are almost always found, e.g., [GPW03, MP07, vEtHSU07, HKW11], it is still possible that discrepancies remain between a specification and a manually derived model. As these transformations are often non-trivial, contain pitfalls, require abstractions or enrichments, and detailed knowledge on the involved formalisms is

necessary, they might introduce gaps and interpretation errors, which can easily stay undetected [ABPV08]. This section describes the possible gaps.

1.3.1 Language Implementation Gap

The first gap concerns the preservation of the behavior when implementing a language. A domain-specific language (DSL) focuses on a particular problem and tends to abstract from irrelevant aspects. For executable languages it may abstract from software implementation details that are added during code generation. Although the technique is applied in industry and promising results are obtained, it nevertheless occurs that generated (template) code is incomplete. Hence, the generated models require post-processing that (i) manually adding vital control information, (ii) wrapping models in "glue code", to match to interfaces of interacting components, or (iii) injecting code to cover (non)-functional aspects, e.g., diagnostic tracing.

Domain-specific languages often use notions from existing formalisms, e.g., [Nie04, ABE10]. Unused parts, parts that do not correspond to desired semantics of a DSL, or concepts that alter over time due to changing requirements, are often subject to semantic changes. Hence, tools and engineers might perceive semantics differently. As the underlying implementation of a language is performed manually, there is no guarantee that the implementation adheres to the specified semantics.

As different languages cover different (design) aspects, potentially spanning over different integration levels, they may influence the interoperability between models and implemented code. All changes made to the models may result in a more time consuming or more difficult integration effort. Consequently, it affects the development and maintainability issues thereafter. So, the lack of integration and interoperability of the underlying languages hinders the use of models as executable contracts between (multi-disciplinary) domains.

1.3.2 Semantic Transformation Gap

The semantic transformation gap relates to the differences introduced by model-to-model transformations. While every environment focuses on a different aspect, (e.g., correctness in the analysis environment, performance in the execution environment, flexibility in the specification environment, and interoperability in the interchange environment) it consequently means that model-to-model transformations change the focus of a model. Hence, in-depth knowledge on the involved languages and their subtleties, and a thorough understanding of the underlying methods and tools are required to perform these transformations. Complementary surveys [CW96, WLBF09] underline these findings.

Syntactic model-transformations are often handcrafted. Therefore, there is always a risk that a syntactic engineered bridge contains undetected errors. Tracking these errors requires at least the same level of expertise as the level required to create a transformation. Moreover, when an error remains undetected, which influences the analysis, it may produce incorrect verdicts.

The semantic expressiveness of a language also contributes to the transformation gap. When models are transformed, it is possible to reason and prove that certain behavior stays preserved. In practice, providing these proofs is hard and time-consuming. Hence, they are neglected and the results of an analysis are validated manually. When model-to-model transformations are performed between informal and formal languages, one should reckon with the interpretation of the semantics for the informal language, as it often comes with ambiguities, sparsely applied abstractions, and ad-hoc transformations.

1.3.3 Cognitive Feedback Gap

The cognitive feedback gap describes the amount of interpretation needed to relate results from an analysis back to a specification model. The cognitive gap occurs when information in the model changes, information is added or is abstracted from, during the transformation (i.e., a focal point change), or are introduced when engineers from different disciplines cooperate. To illustrate the gap, assume a syntactic bridge between the specification and the analysis environment. When we perform the analysis and we observe that a requirement is violated, we need to determine the cause. In most cases, either the specification or the formalization of the requirement is incorrect. However, the cause can also be due to an ill defined transformation or an error in the implementation of the tools that conduct the analysis.

1.4 Syntactically Engineered Models

The first part of this thesis describes four techniques that engineer formal models by using syntactic bridges. The techniques describe and illustrate the bridges from different perspectives. By taking smart design decisions, having detailed knowledge on the involved environments, languages and tools, and relying on the engineer's experience, these bridges informally define and (hopefully) preserve the intended semantics and the relevant properties between environments. The results of the transformations can be used to verify modal properties.

- the modeling of system descriptions (a bridge from the specification environment to the analysis environment - Chapter 3),
- the modeling of an existing implementation (a bridge from the execution environment to the analysis environment - Chapter 4),
- the modeling of a (formal and hybrid) simulation language (a bridge from the interchange environment to the analysis environment - Chapter 5), and
- the round-trip that shows how verification results can be presented in a visual interchange model (a bridge from the analysis environment to the interchange environment - Chapter 6).

1.4.1 Modeling System Descriptions

Modeling a system based on an informal description is typically performed at the start of system development. The route is often used to prototype a system or as an exercise to address the limitations of a formalism [DS09, SRG09].

When we construct a model that possesses the set of desired properties, it ensures that (i) the formalism is suitable for development purposes, and (ii) it increases the chances to actually deliver the system in mind. By means of a comparative case study, we examine in Chapter 3 the suitability of a specification language. For this purpose, we derive formal models from a set of system descriptions. As properties are gradually added, they impose restrictions to the modeled system. The models are constructed and verified using the mCRL2 formalism [GMWU06, GMR+06, GKM+08, Sofb, GMR+09]. Subsequently, we compare the formalism to others for which similar models have been constructed. The case study indicates that the mCRL2 formalism is suitable to specify and verify system designs.

Verification should be performed at the beginning of system development. Even though, it occasionally happens that properties change over time, due to unforeseen (though essential) behavior, mistakes in requirements, restrictions imposed by tools, or customer demand. The implementation of a behavioral model (i.e., the implementation model created in the execution environment) is often written by different engineers by hand. The analysis of a behavioral model (i.e., the behavioral model deduced and used in the analysis environment) is often performed by other engineers. Since both models are created in different environments by different people, they have a higher chance to exhibit different behavior. Consequently, the results obtained during the analysis may not (directly) reflect the actual system's behavior.

1.4.2 Modeling Implementations

Engineering a model from an implementation is a technique that is practiced for two reasons. Firstly, for demonstrating that an implementation complies to a safety standard, e.g., IEC-61508 SIL-4 standard [65A10], or a set of rules and regulations that prevent hazardous or life-threatening situations [WBRG08]. Secondly, when system behavior has become unclear or unpredictable behavior is encountered.

Implementations provide a detailed view on the execution. However directly verifying any requirements is impossible, because the models are too complex to be analyzed. Consequently, abstractions are required to capture the essential behavior of a system and to conduct a meaningful verification.

Chapter 4 presents a case study that derives and analyses a model from software code. The implementation has been used in an industrial application to control the components for a printer that manufactures Printed Circuit Boards. Since the implementation of the actual controller was outsourced, we verified an unrestricted model for violating requirements that could potentially harm the system. The violating requirements are announced to the outsourcing party, along with the traces that led to a violation. In this way, they could implement a hazardous-free controller.

Although the technique has been successfully applied, it has a couple of disadvantages. Firstly, it is a labor-intensive task, because it requires a fully implemented system and models are (almost always) crafted by hand. Secondly, implementations are organic, i.e., they are always subjected to bug-fixes and extensions that provide new features. Thirdly, performing the verification at the (near) end of a development can become costly. Especially when errors are detected that require an iteration in the development trajectory, that could have been prevented by first verifying the design.

1.4.3 Modeling Language Constructs

Models that are constructed through a denotational mapping relate constructs between languages via a function. Depending on the richness of a formalism (expressed by the number of language concepts and the available analytic methods), multiple languages and/or syntactic engineered bridges are required. When a target language defines a superset of concepts, w.r.t. the source language, it is possible to define a single bridge. When a target language defines a subset, often multiple bridges are required.

Chapter 5 translates the modeling language Chi 2.0 to the mCRL2 language. The transformation (indirectly) facilitates verification for the (timed) discrete event part of the Chi 2.0 language. For some examples we show that the translation preserves the intended behavior, by providing some empirical evidence that the translation is correct.

Although both formalisms are process algebra like and express similar behavior, the transformation scheme is complex. It requires detailed knowledge on the involved languages and formalisms. Especially the (small) semantic deviations contribute to a complex transformation. Hence it is not possible to provide mappings for all the constructs. Therefore, this transformation requires human ingenuity to resolve the semantic incompatibility. Since it is hard to guarantee the behavioral equivalence between a source and a target formalism by means of a proof, or for reasons of resources, e.g., time, budget, etc. . . , they are usually omitted.

1.4.4 Disseminating Analysis Results

During system development it is important that knowledge is shared among all involved engineering disciplines. This also holds for verification results. Since models from different environments often have different representations, it is highly unlikely that all of the involved disciplines understand the verification results. Hence, the need arises for solutions that can easily be understood by all.

Chapter 6 promotes a visualization technique that disseminates verification results. With the help of physical models that are actually used within industry, the formal models that describe behavior, and an intermediate model that connects them, we generate animations, that improve the sharing of information and the cooperation within multidisciplinary environments.

Although that such a solution assists in narrowing the gap between the engineering disciplines, it requires interpretive steps, that compose, generate and interpret animations.

1.5 Semantically Engineered Models

The bridges that are presented in the first part of the thesis do not explicitly consider the semantics of a language. Therefore it is possible that unintended behavioral differences are introduced. The second part of the thesis presents a more rigorous approach. It presents a technique for engineering models using the semantics of a language, i.e., the construction of a semantic engineered bridge. Since the bridge is defined both mathematically and computationally, it allows for an automated transformation between different languages, solely based on its formal definition.

In essence, the technique creates a rewrite system for evaluating the formal semantics of a language, i.e., Structural Operational Semantics (SOS). The bridge transforms SOS deduction rules into computational functions. A model combined with the computational functions determines the allowed behavior. As the technique exactly specifies the allowed behavior, it can be used to execute the behavior of a system, reason about behavioral requirements, or it can be applied when prototyping, developing and evaluating formal (domain specific) languages. Since the transformation is specified mathematically, no information is lost or added when transforming models between the environments. To illustrate the semantic transformation route, we describe the following four steps:

- the formalization of an informal domain specific language (Chapter 7),
- the specification of a semantic engineered bridge (Chapter 8),
- a case study that uses the semantic engineered bridge (Chapter 9), and
- a reflection on the semantic engineering technique (Chapter 10).

1.5.1 Formalizing a Behavioral Language

The semantic engineered bridge requires a formal specification language. As many of the available specification languages are still informal, we first show the formalization of an industrial specification language.

The formalization process influences the entire system development. So, it is crucial that all considerations and design decisions are explicitly stated. We illustrate that this is possible and worthwhile for industrial languages.

Chapter 7 describes the formalization of an industrial DSL, called TRECS [Nie04], which is based on the task-resource paradigm. The language uses UML-like activity diagrams to express its behavior. The semantics is informally defined and implicitly implemented in an interpreter. The language is formalized by capturing the essential syntactic notions in its abstract syntax. The behavior of the abstract syntax is

described in SOS. For every syntactic notion we define a set of deduction rules that specify the intended behavior. The process reveals ambiguities and we illustrate how they have been resolved by making small corrections to the language.

1.5.2 Creating a Semantic Engineered Bridge

A semantic engineered bridge can be created for almost any language that describes its semantics in the SOS format. We assume that the language definition is denoted in a Transition System Specification (TSS) [BG96], the set of deduction rules that describe the semantics for a language. Subsequently, we transform a TSS into a Linear Process Specification (LPS).

Chapter 8 provides a detailed description on how to construct models for deduction rules. In short, a sort represents the signature of the abstract syntax, data equations capture the deduction rules, and differently labeled actions describe the different transition relations.

While deduction rules can describe any kind of mathematics, we only specify the semantic bridge for rules that are in De Simone format [dS85]. For these rules we prove that the behavior stays preserved for any translated model.

1.5.3 Applying the Semantic Engineered Bridge

To investigate the suitability of the semantic engineered bridge, we apply the bridge to a formal general purpose language. We define, apply, and subsequently show that it is possible to conduct a meaningful analysis.

Chapter 9 takes the mCRL2 language as input. Since the semantics of the language uses deduction rules that are more expressive than the ones described by the De Simone format, we extend the semantic bridge and feed the semantics of the language to the mCRL2 model checker tools. Hence, we basically dogfood the language.

By means of a set of selected models we investigate that the intended, the specified and the implemented semantics correspond. The validation is performed by generating and comparing state spaces for a variety of models. The exercise improved the (defined) formal semantics of the language and discovered subtle differences in the intended, specified and implemented semantics. As the mCRL2 language has a complexity level that is similar to industrial domain specific languages, we advocate that the exercise can be useful to industry too.

1.5.4 Reflecting on the Semantic Engineered Bridge

Chapter 10 reflects on the semantic engineering approach, by summarizing the lessons learned. We discuss the encountered difficulties and merits. We also illustrate how the results of an analysis can be related to the development models. Furthermore we elaborate on future activities that may be accommodated.

1.6 Structure of the Thesis

The thesis is structured as follows. Chapter 2 contains the preliminaries of the thesis. The preliminaries include the description on Structural Operational Semantics, the mCRL2 language, Linear Process Specifications and the (restricted) modal μ-calculus.

Part I describes syntactic methods for creating formal models, along with a round-trip to visually present verification results. The content is mainly constructed from the work performed in the ITEA2 TWINS Project.

Chapter 3 describes the modeling of three buffers, where every model corresponds to a slightly different system description. The buffers are modeled in the mCRL2 specification language. These models are used to compare the mCRL2 language to other specification languages. This work has been published in [SRG09].

Chapter 4 describes an abstraction technique for modeling code by hiding from the values of variables, after which it is possible to verify safety properties. The content of this chapter originates from [SR09].

Chapter 5 describes the transformation from (a subset of) the (hybrid) specification language Chi 2.0 to the mCRL2 specification language. We formulate a denotational relation that transforms the discrete part of the input language.

Chapter 6 presents a technique that visually combines the results from a formal analysis and physical CAD models. While engineers are often unfamiliar with formal methods, and transformations often contain ad-hoc abstractions, this method provides a clear and solid feedback for multi-disciplinary development teams.

Part II presents a semantic method that transforms a model from a language into a specification language suitable for formal verification. The transformation is performed with the help of the operational semantics of the original language. The target language that has been selected is the mCRL2 language. The content results from work that has been conducted during and after the LithoSysSL Project.

Since many (domain specific) languages are informally defined, Chapter 7 describes the first step of the method, i.e., the formalization of a language. This work has been published in [SWR$^+$11a].

Chapter 8 describes how a formalized language can be converted into a semantic bridge. The approach in this chapter originates from [SRW11a, SRW11b].

Chapter 9 takes the mCRL2 specification language and transforms its operational semantics into an mCRL2 specification. This exercise validates both the semantic method, as well as the correspondence relation between the defined and implemented semantics. The work of this chapter is based on [SRGW11, SRWG12].

Chapter 10 reflects on the semantic method, discusses the lessons learned, illustrates the encountered problems, and elaborates on the possible benefits.

Chapter 11 concludes by discussing our contributions and makes recommendations for practice and further research.

Chapter 2

Preliminaries

2.1 Structural Operational Semantics

Structural Operational Semantics (SOS) defines the possible actions that a piece of syntax is allowed to perform [Plo04]. SOS is typically represented by a Transition System Specification (TSS) [Gro93, BG96]. The syntax for which the semantics is defined, is represented by a signature. A signature fixes the composition operators and their corresponding arities. We assume a set of variables $\mathcal{V}_{\text{SOS}}$ and a set of action labels $\mathcal{A}_{\text{SOS}}$.

Definition 2.1.1 (Signature). A *signature* Σ_{SOS} consists of

- a collection $\mathcal{S}_{\text{SOS}}$ of sort names represented by $S, S_1, \ldots, S_n$,
- a collection of function symbols together with their arities. Let f be a function symbol. Then the arity of a function symbol is denoted by $ar(f)$ and $S_1 \times \cdots \times S_{ar(f)} \to S$ defines the sorts of the function symbol. The domain $S_1 \times \cdots \times S_{ar(f)}$ may be empty.

Definition 2.1.2 (Term). The collection of (open) *terms* over signature Σ_{SOS}, denoted $\mathcal{T}(\Sigma)$, is the smallest set such that

- a variable $x_S \in \mathcal{V}^S_{\text{SOS}}$ is a term of sort S, where $\bigcup_{s \in \mathcal{S}_{\text{SOS}}} \mathcal{V}^s_{\text{SOS}} = \mathcal{V}_{\text{SOS}}$, and
- $f(t_1, \ldots, t_n)$ is a term of sort S, if $t_1, \ldots, t_n$ are terms, where t_i is a term of sort S_i and $f \in \Sigma_{\text{SOS}}$ is an n-ary function symbol of sort $S_1 \times \cdots \times S_n \to S$.

The set of *closed terms* over signature Σ_{SOS}, denoted $\mathcal{C}(\Sigma)$, is the set of all terms over Σ_{SOS} in which no variables occur. Variables that occur in a term p are retrieved by the function $vars : \mathcal{T}(\Sigma) \to 2^{\mathcal{V}_{\text{SOS}}}$, denoted $vars(p)$. For any variable x_S, Λ^S_{SOS} denotes the set of allowable values corresponding to the variable x of sort S.

Definition 2.1.3 (Valuation). A *valuation* $\sigma : \mathcal{V}_{\text{SOS}} \to \Lambda_{\text{SOS}}$ is a partial function from variables of sort S to values of the same sort. We assume that valuations are closed, i.e., every (defined) variable maps to a value Λ_{SOS}. For every element from Λ_{SOS} we assume there exists a syntactic representation in $\mathcal{T}(\Sigma)$.

Definition 2.1.4 (Transition Formula). Let $p, p' \in \mathcal{T}(\Sigma)$ be terms, let $l \in \mathcal{A}_{\text{SOS}}$, and let $\sigma, \sigma' : \mathcal{V}_{\text{SOS}} \to \Lambda_{\text{SOS}}$, then a *transition formula* over Σ is of the form $(p, \sigma) \xrightarrow{l} (p', \sigma')$.

Definition 2.1.5 (Transition System Specification). A *Transition System Specification* (TSS) [GV92, Gro93, BG96] denotes a set of deduction rules. It is defined by a tuple $(\Sigma_{\text{SOS}}, \mathcal{D}_{\text{SOS}})$, where Σ_{SOS} is a signature and $\mathcal{D}_{\text{SOS}}$ is a set of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of transition formulas over Σ, called the set of *premises*, and C is a transition formula, called the *conclusion*. To derive the conclusion, and perform an action, all premises need to be satisfied. The formal definition of a TSS can be found in [GV92].

2.2 The mCRL2 Language

The micro Common Representation Language 2 (mCRL2) is an action based specification language intended for the description and verification of the behavior of distributed systems, parallel computer programs and protocols. The language is supported by a toolset that enables simulation, state-space visualization, behavioral reduction techniques and verification of software requirements [GMWU06, GMR+06, GKM+08, Sofb].

The mCRL2 language originates from the Algebra of Communicating Processes (ACP) [BK85]. After extending ACP with abstract data types the authors of [GR01] defined the μCRL language, supported by the μCRL toolset [BFG+01]. With the help of this language they modeled and verified a variety of systems. Motivated by practical experiences and new theoretical insights [GMWU06], the mCRL2 specification language emerged as the successor of μCRL. The key contributions of the new language include the addition of commonly used abstract data types, behavioral constructs to model true concurrency with the help of multi-actions, and the specification of (real) timed processes.

The language consists of a behavioral specification part and a data specification part. The behavioral specification part is commonly used to express the modeled behavior of a system. The data specification part is commonly used to express computations, based on higher-order abstract equational data types. It contains quantifiers, (unbounded) integers, (infinite) sets and bags, structured types, lists and real numbers. These concepts are set up to be as close as possible to their mathematical counterparts. This means that the language is very expressive and it is easy to write down undecidable properties. For the decidable part advanced algorithms have been devised, such as just-in-time compiling rewriting [Wee07], that provide the tools high performance rewriting despite the generality of data types.

This section describes the syntactic elements of the mCRL2 language and the associated formal semantics in SOS.

2.2.1 Syntactic Concepts

An mCRL2 specification consists of a data specification and a process specification.

Data Specification

We assume that a set of sorts $\mathcal{S}^{\text{mCRL2}}$, a set of constructors $\mathcal{C}^{\text{mCRL2}}$ and a set of mappings $\mathcal{M}^{\text{mCRL2}}$ are available.

Definition 2.2.1 (Signature). Let $\mathcal{S}^{\text{mCRL2}}$ be a set of sorts, $\mathcal{C}_{\mathcal{S}}^{\text{mCRL2}}$ a set of function symbols over $\mathcal{S}^{\text{mCRL2}}$ called constructors, and $\mathcal{M}_{\mathcal{S}}^{\text{mCRL2}}$ a set of function symbols over $\mathcal{S}^{\text{mCRL2}}$ called mappings. We call the triple $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ a signature.

Definition 2.2.2 (Constructor sort). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature. Sort $S \in \mathcal{S}^{\text{mCRL2}}$ is a *constructor sort* if there exists a constructor function declaration $f : S_1 \times \cdots \times S_n \to S \in \mathcal{C}^{\text{mCRL2}}$.

We assume that a signature is well typed.

Definition 2.2.3 (Well-typed signature). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature. Then signature Σ^{mCRL2} is well-typed iff:

- $\mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cap \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}} = \emptyset$,
- $\mathbb{B}$ is a sort, with exactly the constructors *true*:$\mathbb{B}$ and *false*:$\mathbb{B}$, and
- Constructor sorts are *syntactically non empty*. A sort D is defined to be *syntactically non empty* iff there is a constructor $f{:}D_1 \times \cdots \times D_n \to D \in \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} (n \geq 0)$ such that for all $1 \leq i \leq n$ if D_i is a constructor sort, D_i is also syntactically non empty, and $D \neq D_i$.

Definition 2.2.4 (Data expressions). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}}, \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}})$ be a signature. Let $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$ be a set of $\mathcal{S}$-typed variable symbols. Then, we inductively define typed data expressions (over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$) as follows:

- every variable symbol $x{:}D \in \mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$ of sort D is a data expression of sort D.
- every function symbol $f{:}D \in \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}}$ is a data expression of sort D.
- Let p be a data expression of sort $D_1 \times \cdots \times D_n \to D$ and for $1 \leq i \leq n$ let p_i be a data expression of sort D_i, then $p(p_1, \ldots, p_n)$ is a data expression of sort D.
- For $1 \leq i \leq n$, if $x_i{:} \in \mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$ or $x_i \notin (\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}})$, and p is a data expression of sort D over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$, then $\lambda_{x_1:D_1,\ldots,x_n:D_n} p$ is a data expression of sort $D_1 \times \cdots \times D_n \to D$.

- For $1 \leq i \leq n$, if $x_i : \in \mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$ or $x_i \notin (\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}})$ and p is a data expression of sort $\mathbb{B}$ over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$, then $\exists_{x_1:D_1,\cdots,x_n:D_n} p$ is a data expression of sort $\mathbb{B}$.
- For $1 \leq i \leq n$, if $x_i : \in \mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$ or $x_i \notin (\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}})$, and p is a data expression of sort $\mathbb{B}$ over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$, then $\forall_{x_1:D_1,\cdots,x_n:D_n} p$ is a data expression of sort $\mathbb{B}$.
- For $1 \leq i \leq n$, let p_i be a data expression of sorts D_i over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}}$, $x_i{:}D_i \notin (\mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}})$, and let p be a data expression of sort D over $\mathcal{X}_{\mathcal{S}}^{\text{mCRL2}} \cup \{x_i{:}D_i | 1 \leq i \leq n\}$ then p **whr** $x_1 = p_1, \ldots, x_n = p_n$ **end** is a data expression of sort D.

Definition 2.2.5 (Data specification). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a well-typed signature. Then the tuple $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ is a *data specification*, in which E is a set of *conditional equations*. Every equation in E is a pair $\langle \mathcal{X}^{\text{mCRL2}}, c \rightarrow p_1 = p_2 \rangle$. Here $\mathcal{X}^{\text{mCRL2}}$ is a set of variable declarations and $c{:}\mathbb{B}$, $p_1{:}D$ and $p_2{:}D$ are data expressions, where $D \in \mathcal{S}^{\text{mCRL2}}$.

Process Specification

We assume that a set of action labels $\mathcal{A}^{\text{mCRL2}}$ is available. We use $\overrightarrow{\mathcal{A}^{\text{mCRL2}}}$ to denote a vector of action labels.

Definition 2.2.6 (Action declaration). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature, let $\mathcal{A}^{\text{mCRL2}}$ be a set of action labels, and for $1 \leq i \leq n$ let $D_i \in \mathcal{S}^{\text{mCRL2}}$, and let $a \in \mathcal{A}^{\text{mCRL2}}$, then an *action declaration* is an expressions of the form $a{:}D_1 \times \cdots \times D_n$.

An action a with data expressions $d_1, \ldots, d_n$, denoted $\overrightarrow{d}$, is written as $a(\overrightarrow{d})$. Actions may be declared without any sorts, denoting actions without any data parameters. These actions are written as a.

All actions that are specified inside a *process specification* (Definition 2.2.11) are declared by an *action declaration*. We assume that all actions that occur in a *process expression* (Definition 2.2.10) are declared.

Definition 2.2.7 (Syntactic multi-action). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature and let $\mathcal{A}^{\text{mCRL2}}$ be a set of action labels. A syntactic multi-action represents a collection of actions that are specified to occur at the same time instant. Syntactic multi-actions have the following BNF grammar:

$$\alpha ::= \tau \mid a(\overrightarrow{d}) \mid \alpha | \alpha,$$

The terminal τ represents an empty multi-action. The terminal $a(\overrightarrow{d})$ represents an action, where $a \in \mathcal{A}^{\text{mCRL2}}$ denotes an action label and $\overrightarrow{d} : \overrightarrow{D}$ a vector of data expressions such that $D_i \in \mathcal{S}^{\text{mCRL2}}$ for each $D_i \in \overrightarrow{D}$. The non-terminal α represents a syntactic multi-action. The syntactic multi-action $\alpha | \alpha'$ consists of the actions from both the syntactic multi-actions α and α'.

Definition 2.2.8 (Action declaration). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature. An *action declaration* is an expression of the form $a : D_1 \times \cdots \times D_n$ where $n \geq 0$ and all are sorts D_i are taken from $\mathcal{S}^{\text{mCRL2}}$.

Definition 2.2.9 (Process expression). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature, such that the tuple $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ is a *data specification*. Process expressions are expressions with the following syntax:

$$p ::= \delta \mid \alpha \mid p + p \mid p \cdot p \mid c \rightarrow p \mid c \rightarrow p \diamond p \mid \sum_{v:D} p \mid p^{\triangleleft} t \mid t \gg p \mid p \ll p \mid$$
$$p \| p \mid p \lfloor\!\lfloor p \mid p | p \mid \Gamma_C(p) \mid \nabla_V(p) \mid \partial_B(p) \mid \rho_R(p) \mid \tau_I(p) \mid \Upsilon_U(p) \mid$$
$$X(v_1{=}d_1, \ldots, v_n{=}d_n)$$

The process terms that are colored black belong to the untimed fragment of the mCRL2 language. If we consider the untimed fragment of the mCRL2 language, we only consider these process terms.

In the above BNF, p denotes a process term, α is a syntactic multi-action, $c \in \mathbb{B}$ is a Boolean data-expression, v, $v_1, \ldots, v_n \in \mathcal{X}^{\text{mCRL2}}$ $(n \geq 0)$ are variables, $D \in \mathcal{S}^{\text{mCRL2}}$ is a sort, $t \in \mathbb{R}^{\geq 0}$ is a positive Real data-expression, $C \subseteq \mathcal{A}^{\text{mCRL2}} \times \cdots \times \mathcal{A}^{\text{mCRL2}} \rightarrow \mathcal{A}^{\text{mCRL2}}$ a set of communications, $V \subseteq \mathcal{A}^{\text{mCRL2}} \times \cdots \times \mathcal{A}^{\text{mCRL2}}$ a set of multi-action labels, $B \subseteq \mathcal{A}^{\text{mCRL2}}$, $I \subseteq \mathcal{A}^{\text{mCRL2}}$ and $U \subseteq \mathcal{A}^{\text{mCRL2}}$ are sets of action labels, $R \subseteq \mathcal{A}^{\text{mCRL2}} \rightarrow \mathcal{A}^{\text{mCRL2}}$ is a set of renamings, and $d_1, \ldots, d_n$ are data expressions.

For processes, $p + q$ denotes the non-deterministic choice, $p \cdot q$ denotes the sequential composition, $c \rightarrow p$ denotes the conditional if-then execution, $c \rightarrow p \diamond p$ denotes the conditional if-then-else execution, $\sum_{v:D} p$ denotes the non-deterministic choice over the domain of D by selecting a value for variable v, $p^{\triangleleft} t$ denotes that process p has to be executed at time t, $t \gg p$ denotes the auxiliary initialization operator saying that process p must start after time t, $p \ll p$ describes the part of the left process p that can happen before the right process p must perform an action, $p \parallel q$ denotes the parallel composition, $p \lfloor\!\lfloor q$ denotes the left merge composition and $p|q$ denotes the synchronized composition. The process expression $\nabla_A(p)$ allows only the multi-actions from the set A of multi-action labels occurring in process p. $\partial_B(p)$ blocks all actions in process p for which the corresponding action labels occur in the set of action labels B. $\Gamma_C(p)$ applies the communications described by C to process p. $\tau_I(p)$ hides all actions in process p for which the corresponding action labels occur in set of action labels I. $\Upsilon_U(p)$ pre-hides all actions in process p for which the corresponding action labels occur in set of action labels U. X is a recursion variable, $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ is a process reference to a *process equation* of the form $X(v_1{:}D_1, \ldots, v_n{:}D_n) = p$, i.e., the process $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ that behaves as p where the occurrences of $v_1, \ldots, v_n$ are substituted with $d_1, \ldots, d_n$. Alternatively, if we assume that all substitutions are performed for the corresponding process parameters $X(v_1{=}d_1, \ldots, v_n{=}d_n)$ it can also be expressed as $X(d_1, \ldots, d_n)$.

Definition 2.2.10 (Process equation). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature. A *process equation* is an expression of the form $X(v_1{:}D_1, \ldots, v_n{:}D_n) = p$, p is a process expression, and for $(1 \leq i \leq n)$, v_i are variables of sort D_i from $\mathcal{S}^{\text{mCRL2}}$.

Definition 2.2.11 (Process specification). A *process specification* is a five tuple $PS = (\mathcal{D}^{\text{mCRL2}}, AD, PE, p, \mathcal{X}^{\text{mCRL2}})$ where

- $\mathcal{D}^{\text{mCRL2}}$ is a data specification,
- AD is an action declaration,
- PE is a set of process equations,
- p is a process expression, and
- $\mathcal{X}^{\text{mCRL2}}$ is a set of global variables.

For reasons of simplicity, we assume that all process specifications and their underlying components are well-typed as described in [KR11].

2.2.2 Semantic Concepts

Definition 2.2.12 (Applicative $\mathcal{D}^{\text{mCRL2}}$-structure). Let $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ be a data specification. Then the collection of nonempty sets $\{M_D | D \in \mathcal{S}_{\mathcal{S}}^{\text{mCRL2}}\}$ is an *applicative $\mathcal{D}^{\text{mCRL2}}$-structure* iff:

- $\mathcal{D}_{\mathbb{B}}^{\text{mCRL2}}$ is a set with two elements, denoted by **true** and **false**, for which **true** $\neq$ **false** holds.
- $D \in \mathcal{S}^{\text{mCRL2}}$ and D is not a function symbol, then M_D is a nonempty set.
- $D = D_1 \times \cdots \times D_n \to D'$, then M_D is the set of all functions from $M_{D_1} \times \cdots \times M_{D_n} \to M_{D'_n}$.

Definition 2.2.13 (Valuation). Let $\sigma{:}\mathcal{X}^{\text{mCRL2}} \to \bigcup_{D \in \mathcal{S}_{\mathcal{S}}^{\text{mCRL2}}} M_D$, then σ is a *valuation* if $\sigma(v) \in M_D$ holds for all $v{:}\mathcal{X}_D^{\text{mCRL2}}$. We write $\sigma[v_i \mapsto w_i]_{1 \le i \le n}$ (or $\sigma[\overrightarrow{v} \mapsto \overrightarrow{w}]$) for a valuation σ with for $1 \le i \le n$ the function updates $[v_i \mapsto w_i]$, that maps all variables according to σ, except for the variables v_i $(1 \le i \le n)$. These variables are mapped to the corresponding values of w_i.

Definition 2.2.14 (Semantic interpretation). Let $\Sigma^{\text{mCRL2}} = (\mathcal{S}^{\text{mCRL2}}, \mathcal{C}^{\text{mCRL2}}, \mathcal{M}^{\text{mCRL2}})$ be a signature, let $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ be a data specification and let σ be a data valuation. The semantic interpretation function $\{[\cdot]\}^{\sigma}$ on a data expression is defined through:

- $\{[v]\}^{\sigma} = \sigma(v)$ for every variable $v \in \mathcal{X}_D^{\text{mCRL2}}$ $(D \in \mathcal{S}^{\text{mCRL2}})$.
- $\{[f]\}^{\sigma} = \{[f]\}$ for every function symbol $f \in \mathcal{C}_{\mathcal{S}}^{\text{mCRL2}} \cup \mathcal{M}_{\mathcal{S}}^{\text{mCRL2}}$ and $\{[f]\} \in M_D$.
- $\{[p(p_1, \ldots, p_n)]\}^{\sigma} = \{[p]\}^{\sigma}(\{[p_1]\}^{\sigma}, \ldots, \{[p_n]\}^{\sigma})$
- $\{[\lambda_{x_1:D_1,\ldots,x_n:D_n} p]\}^{\sigma} = f$ where function $f{:}M_{D_1} \times \cdots \times M_{D_n} \to D$ satisfies $f(d_1, \ldots, d_n) = \{[p]\}^{\sigma[x_i \mapsto d_i]_{1 \le i \le n}}$ for all $d_i{:}M_D$.

- $\{[\forall_{x_1:D_1,\ldots,x_n:D_n} p]\}^\sigma = \textbf{true}$ iff for all $d_i \in M_D$ it holds that $\{[p]\}^{\sigma[x_i \mapsto d_i]_{1\leq i\leq n}} = \textbf{true}$.
- $\{[\exists_{x_1:D_1,\ldots,x_n:D_n} p]\}^\sigma = \textbf{true}$ iff for some $d_i \in M_D$ it holds that $\{[p]\}^{\sigma[x_i \mapsto d_i]_{1\leq i\leq n}} = \textbf{true}$.
- $\{[p \textbf{ whr } x_1{=}p_1, \ldots, x_n{=}p_n \textbf{ end}]\}^\sigma = \{[p]\}^{\sigma[x_i \mapsto \{[p_i]\}^\sigma]_{1\leq i\leq n}}$.

Definition 2.2.15 ($\mathcal{D}^{\text{mCRL2}}$-model interpretation). Let $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ be a data specification, let σ be a data valuation and let $\{M_D | D{\in}\mathcal{S}^{\text{mCRL2}}\}$ be a $\mathcal{D}^{\text{mCRL2}}$-model where M_D is the domain of sort D. The interpretation of a $\mathcal{D}^{\text{mCRL2}}$*-model* is defined through $\{[\cdot]\}$:

- for every equation $c{\rightarrow}p_1{=}p_2 \in E_\mathcal{S}$ it holds that if $\{[c]\}^\sigma = \textbf{true}$ then $\{[p_1]\}^\sigma = \{[p_2]\}^\sigma$ for every valuation σ.
- $\{[\mathit{true}]\}^\sigma = \textbf{true}$ and $\{[\mathit{false}]\}^\sigma = \textbf{false}$ for every valuation σ.
- If a basic sort D is a constructor sort (i.e., there is a constructor $f \in \mathcal{C}_\mathcal{S}^{\text{mCRL2}}$ of sort $D_1 \times \cdots \times D_n \rightarrow D$), then every element $d \in M_D$ is a constructor element. A constructor element is inductively defined by:
  - An element $d \in M_D$ is a constructor element, if D is a constructor sort and a constructor function $f \in \mathcal{C}_\mathcal{S}^{\text{mCRL2}}$ of sort $D_1 \times \cdots \times D_n \rightarrow D$ exists such that $d = \{[f]\}(e_1, \ldots, e_n)$ where for each $1 \leq i \leq n$, e_i is either a constructor element of sort D_i, or
  - sort D_i is not a constructor sort.

Definition 2.2.16 (Semantic multi-action). Let $\mathcal{D}^{\text{mCRL2}} = (\Sigma^{\text{mCRL2}}, E)$ be a data specification, $\{[\cdot]\}^\sigma$ an interpretation, E a set of data equations, $a \in \mathcal{A}^{\text{mCRL2}}$ and $w_1 : M_{D_1}, \ldots, w_n : M_{D_n}$ are values. Let the interpretation of any syntactic multi-action $[\![\cdot]\!]^\sigma$ on α, β be inductively defined for any data-valuation σ by:

- $[\![\tau]\!]^\sigma = \tau$.
- $[\![a(w_1, \ldots, w_n)]\!]^\sigma = a(\{[w_1]\}^\sigma, \ldots, \{[w_n]\}^\sigma)$.
- $[\![\alpha|\beta]\!]^\sigma = [\![\alpha]\!]^\sigma | [\![\beta]\!]^\sigma$.

A semantic action that has no data parameters, i.e., $a()$, may be written as a.

Definition 2.2.17 (Semantic multi-action equivalence class). Let α, β be semantic multi-actions, then the semantic multi-action equivalence relation is defined as the smallest equivalence relation $\sim$ that satisfies:

$$\begin{cases} \alpha \mid \tau \sim \alpha \\ \alpha \mid \beta \sim \beta \mid \alpha \\ (\alpha \mid \beta) \mid \gamma \sim \alpha \mid (\beta \mid \gamma) \end{cases}$$

The equivalence class with respect to $\sim$ of a multi-action α is denoted by a $\sim$ subscript:

$$\alpha_\sim = \{\beta :: \beta \sim \alpha\}$$

Furthermore, we define a function, denoted $(\cdot)_\sim$, that merges separate equivalence classes into a new equivalence class. Let $a \in \mathcal{A}^{\text{mCRL2}}$ and let $w_1, \ldots, w_n$ denote values, then the function is defined as:

$$\begin{cases} (\tau_\sim)_\sim = \tau_\sim \\ (a(w_1, \ldots, w_n)_\sim)_\sim = a(w_1, \ldots, w_n)_\sim \\ (\alpha_\sim \mid \beta_\sim)_\sim = (\alpha \mid \beta)_\sim \end{cases}$$

The semantics of the processes are defined using inference rules. These rules extract information from semantic multi-action equivalence classes.

Definition 2.2.18 (Functions on semantic multi-action equivalence classes).

- $\alpha_\sim^{\{\}}$ is the set of all action labels that occurs in the semantic multi-action equivalence class $\alpha_\sim$. The function is inductively defined as:
  - $\tau_\sim^{\{\}} = \emptyset$
  - $a(w_1, \ldots, w_n)_\sim^{\{\}} = \{a\}$
  - $\alpha|\beta_\sim^{\{\}} = \alpha_\sim^{\{\}} \cup \beta_\sim^{\{\}}$
- $\underline{\alpha_\sim}$ denotes the semantic multi-action equivalence class $\alpha_\sim$ from which all data parameters are removed. The function inductively defined as:
  - $\underline{\tau_\sim} = \tau_\sim$
  - $\underline{a(w_1, \ldots, w_n)_\sim} = a_\sim$
  - $\underline{(\alpha \mid \beta)_\sim} = (\underline{\alpha_\sim} \mid \underline{\beta_\sim})_\sim$
- Let R be a set of renamings. Then the function $R \bullet (\alpha_\sim)$ denotes the renaming on a semantic multi-action equivalence class. Here the action labels are renamed according to the renamings of R. The function is inductively defined as:
  - $R \bullet (\tau_\sim) = \tau_\sim$
  - $R \bullet (a(w_1, \ldots, w_n)_\sim) = \begin{cases} b(w_1, \ldots, w_n)_\sim & \text{if } a \rightarrow b \in R \text{ for some } b \\ a(w_1, \ldots, w_n)_\sim & \text{if } a \rightarrow b \notin R \text{ for all } b \end{cases}$
  - $R \bullet ((\alpha|\beta)_\sim) = (R \bullet (\alpha_\sim) \mid R \bullet (\beta_\sim))_\sim$
- $\theta_I(\alpha_\sim)$ hides the actions in a semantic multi-action equivalence class $\alpha_\sim$ for which the corresponding labels occur in I. The function is inductively defined as:
  - $\theta_I(\tau_\sim) = \tau_\sim$

- $\theta_I(a(w_1,\ldots,w_n)_\sim) = \begin{cases} \tau_\sim & \text{if } a \in I \\ a(w_1,\ldots,w_n)_\sim & \text{if } a \notin I \end{cases}$
- $\theta_I((\alpha|\beta)_\sim) = (\theta_I(\alpha_\sim) \mid \theta_I(\beta_\sim))_\sim$

- $\eta_U(\alpha_\sim)$ prehides the actions in a semantic multi-action equivalence class $\alpha_\sim$ for which the corresponding labels occur in U. Prehiding is accomplished by removing all data parameters and relabeling the action label to *int*. The function is inductively defined as:
  - $\eta_U(\tau_\sim) = \tau_\sim$
  - $\eta_U(a(w_1,\ldots,w_n)_\sim) = \begin{cases} int_\sim & \text{if } a \in U \\ a(w_1,\ldots,w_n)_\sim & \text{if } a \notin U \end{cases}$
  - $\eta_U((\alpha|\beta)_\sim) = (\eta_U(\alpha_\sim) \mid \eta_U(\beta_\sim))_\sim$
- Communication is defined using γ_C. Let $c_i \equiv c_i^1|\ldots|c_i^{m_i}$, then we define the communication function $C = \{c_1 \to c_1', \ldots, c_n \to c_n'\}$. The set of synchronizing actions are represented by $c_1,\ldots,c_n \in \mathcal{A}^{\text{mCRL2}} \times \cdots \times \mathcal{A}^{\text{mCRL2}}$, and the communication results are represent by $c_1',\ldots,c_n' \in \mathcal{A}^{\text{mCRL2}}$. The specification assumes that all action labels of a domain of a single communication function are pairwise disjoint, i.e.,
$$\forall_{I,J \in dom(C)} I \neq J \Rightarrow f_\emptyset(I,J)$$
where:
$$\begin{aligned} f_\emptyset([],J) &= true \\ f_\emptyset(x \triangleright xs, J) &= x \notin J \wedge f_\emptyset(xs,J) \end{aligned}$$
Communication takes place over a semantic multi-action equivalence class, and only applies it to those actions for which the arguments have the same semantic logic equivalent values. Let $\overrightarrow{w} : \overrightarrow{M_D}$, let $c_i(\overrightarrow{w}) \equiv c_i^1(\overrightarrow{w}) \mid \ldots \mid c_i^{m_i}(\overrightarrow{w})$, let $\alpha_\sim \sqsubseteq \beta_\sim$ be the inclusion of $\alpha_\sim$ in $\beta_\sim$, and let $\alpha_\sim \backslash \beta_\sim$ be the removal of the semantic actions of $\beta_\sim$ from $\alpha_\sim$. We specify the communication $\gamma_C(\alpha_\sim)$ as:
$$\gamma_C(\alpha_\sim) = \begin{cases} (c_i'(\overrightarrow{w}) \mid \gamma_C(\alpha_\sim \backslash c_i(\overrightarrow{w})_\sim))_\sim & \text{if } \exists_{\overrightarrow{w}} \exists_{c_i} (c_i \to c_i') \in C \\ & \quad \wedge\, c_i(\overrightarrow{w})_\sim \sqsubseteq \alpha_\sim \\ \alpha_\sim & \text{otherwise} \end{cases}$$
The function defines the communication recursively. Intuitively, if there exists a set of actions labels (obtained after the data elimination of a semantic multi-action equivalence class) that occurs in the domain of communication function, it is replaced by the corresponding action label from the image with the appropriate data values, and the communication function is again applied to the remainder of multi-action equivalence class. The communication returns the input, when no instances are found. Observe that all communication functions are orthogonal, because the synchronization domains of a communication function C are all pairwise disjoint. Hence, the order in which the functions are applied does not affect the outcome of the communication function C.

2.2.3 mCRL2's Structural Operational Semantics

Given a data specification and a process expression, we express the semantics of the process expression through a transition system. The way in which a process expression relates to a transition system is described via deduction rules. The rules that relate to the timed mCRL2 fragment are shown in gray.

Definition 2.2.19 (Semantics of a process). Let $PS = (\mathcal{D}^{\text{mCRL2}}, AD, PE, p, \mathcal{X}^{\text{mCRL2}})$ be a process specification. Let $\{M_D | D \in \mathcal{S}^{\text{mCRL2}}\}$ be a $\mathcal{D}^{\text{mCRL2}}$-model where M_D is the domain of sort D, $\{[\cdot]\}^{\sigma}$ a semantic-interpretation and σ a data valuation. We define the semantics for a process specification PS given $\mathcal{A}^{\text{mCRL2}}$ and σ_0 as the initial data valuation, by the transition system $A = (S, Act, \longrightarrow, \rightsquigarrow, s_0, T)$:

- The states S contain all pairs (p', σ') for process expressions p' and valuations σ'. There is one special termination state, denoted by the $\checkmark$ predicate.
- A label denotes a semantic multi-action equivalence class in Act.
- The transitions are inductively defined by the operational rules in Tables 2.1, 2.2, 2.3, 2.4, 2.5, 2.6 and 2.7. These rules describe the semantics of the mCRL2 language. The transition relation is denoted by $(p', \sigma) \xrightarrow{m}_t (p'', \sigma'), (p', \sigma) \xrightarrow{m}_t \checkmark \subseteq S \times Act \times \mathbb{R} \times S$. Note that some (parts) of the deduction rules are colored gray. The gray colored deduction rules belong to the timed fragment of the mCRL2 language. If we consider the untimed fragment we abstract from the gray colored parts.
- The idle relation expresses that a state can idle up to and including time $t \in \mathbb{R}^{\geq 0}$, denoted by $p \rightsquigarrow_t$ in the deduction rules.
- The initial state s_0 corresponds to (p, σ_0).
- $T \subseteq S$ is the set of *terminating states*.

For reasons of completeness we specify all of mCRL2's deduction rules. The untimed deduction rules are explicitly used in Chapter 9 when modeling the semantics of the mCRL2 language. Implicitly, the entire set of rules is considered when we take design decisions, specify (timed) models, or provide modeling constructs in all other chapters.

2.3 Linear Process Specifications

A Linear Process Specification (LPS) is a symbolic representation for capturing (possibly infinite) Labeled Transition Systems (LTS). Informally, an LPS consists of a signature, a collection of variable declarations, a collection of data equations, a collection of action declarations, a linear process equation, and an initialization. A (full) formal definition of an LPS and its components is found in [GMR+06].

$(Ma)\dfrac{}{(\alpha,\sigma)\xrightarrow{[\![\alpha]\!]^{\sigma}_{\sim}}_{t}\checkmark}$	$(Ma^{t})\dfrac{}{(\alpha,\sigma)\rightsquigarrow_{t}}$	$(Delta^{t})\dfrac{}{(\delta,\sigma)\rightsquigarrow_{t}}$
$(Alt_1)\dfrac{(p,\sigma)\xrightarrow{m}_{t}\checkmark}{(p+q,\sigma)\xrightarrow{m}_{t}\checkmark}$	$(Alt_2)\dfrac{(p,\sigma)\xrightarrow{m}_{t}(p',\sigma')}{(p+q,\sigma)\xrightarrow{m}_{t}(p',\sigma')}$	$(Alt^{t}_1)\dfrac{(p,\sigma)\rightsquigarrow_{t}}{(p+q,\sigma)\rightsquigarrow_{t}}$
$(Alt_3)\dfrac{(q,\sigma)\xrightarrow{m}_{t}\checkmark}{(p+q,\sigma)\xrightarrow{m}_{t}\checkmark}$	$(Alt_4)\dfrac{(q,\sigma)\xrightarrow{m}_{t}(q',\sigma')}{(p+q,\sigma)\xrightarrow{m}_{t}(q',\sigma')}$	$(Alt^{t}_2)\dfrac{(q,\sigma)\rightsquigarrow_{t}}{(p+q,\sigma)\rightsquigarrow_{t}}$
$(Seq_1)\dfrac{(p,\sigma)\xrightarrow{m}_{t}\checkmark}{(p\cdot q,\sigma)\xrightarrow{m}_{t}(t\gg q,\sigma)}$	$(Seq_2)\dfrac{(p,\sigma)\xrightarrow{m}_{t}(p',\sigma')}{(p\cdot q,\sigma)\xrightarrow{m}_{t}(p'\cdot q,\sigma')}$	$(Seq^{t})\dfrac{(p,\sigma)\rightsquigarrow_{t}}{(p\cdot q,\sigma)\rightsquigarrow_{t}}$
$(Cond_1)\dfrac{\{[b]\}^{\sigma}=\mathbf{true}\quad (p,\sigma)\xrightarrow{m}_{t}\checkmark}{(b\rightarrow p,\sigma)\xrightarrow{m}_{t}\checkmark}$	$(Cond_2)\dfrac{\{[b]\}^{\sigma}=\mathbf{true},\quad (p,\sigma)\xrightarrow{m}_{t}(p',\sigma')}{(b\rightarrow p,\sigma)\xrightarrow{m}_{t}(p',\sigma')}$	$(Cond^{t})\dfrac{\{[b]\}^{\sigma}=\mathbf{true},\quad (p,\sigma)\rightsquigarrow_{t}}{(b\rightarrow p,\sigma)\rightsquigarrow_{t}}$
$(Cond'_1)\dfrac{\{[b]\}^{\sigma}=\mathbf{true},\quad (p,\sigma)\xrightarrow{m}_{t}\checkmark}{(b\rightarrow p\diamond q,\sigma)\xrightarrow{m}_{t}\checkmark}$	$(Cond'_2)\dfrac{\{[b]\}^{\sigma}=\mathbf{true},\quad (p,\sigma)\xrightarrow{m}_{t}(p',\sigma')}{(b\rightarrow p\diamond q,\sigma)\xrightarrow{m}_{t}(p',\sigma')}$	$(Cond^{t\,\prime}_1)\dfrac{\{[b]\}^{\sigma}=\mathbf{true},\quad (p,\sigma)\rightsquigarrow_{t})}{(b\rightarrow p\diamond q,\sigma)\rightsquigarrow_{t}}$
$(Cond'_3)\dfrac{\{[b]\}^{\sigma}=\mathbf{false},\quad (q,\sigma)\xrightarrow{m}_{t}\checkmark}{(b\rightarrow p\diamond q,\sigma)\xrightarrow{m}_{t}\checkmark}$	$(Cond'_4)\dfrac{\{[b]\}^{\sigma}=\mathbf{false},\quad (q,\sigma)\xrightarrow{m}_{t}(q',\sigma')}{(b\rightarrow p\diamond q,\sigma)\xrightarrow{m}_{t}(q',\sigma')}$	$(Cond^{t\,\prime}_2)\dfrac{\{[b]\}^{\sigma}=\mathbf{false},\quad (q,\sigma)\rightsquigarrow_{t})}{(b\rightarrow p\diamond q,\sigma)\rightsquigarrow_{t}}$

Table 2.1 SOS deduction rules for the basic operators

An LPS is a restricted mCRL2 specification. That is, the process specification is defined through a single process equation that represents the behavior of the mCRL2 specification. An explanation that is analogue to the relation between an mCRL2 specification and an LPS can be found in [Use02], which describes the relation between an μCRL specification and an LPS. The signature, variables, data equations and action declarations of the LPS respectively correspond to their counterparts in the mCRL2 language.

Definition 2.3.1 (Linear Process Equation). A linear process equation (LPE) is a process of the following form

$$\begin{aligned} X(d{:}D) \quad &= \quad \sum_{i\in I}\sum_{e_i:E_i} c_i(d,e_i)\rightarrow\alpha_i(d,e_i)\text{‹}t_i(d,e_i)\cdot X(g_i(d,e_i)) \\ &+ \quad \sum_{j\in J}\sum_{e_j:E_j} c_j(d,e_j)\rightarrow\alpha_{\delta j}(d,e_j)\text{‹}t_j(d,e_j) \end{aligned}$$

where $i\in I$ and $j\in J$ are *meta-level* variables denoting the two finite index sets, and

$$(Sum_1)\frac{w \in M_D,\ (p, \sigma[v \mapsto w]) \xrightarrow{m}_t \checkmark}{(\sum_{v:D} p, \sigma) \xrightarrow{m}_t \checkmark} \qquad (Sum_2)\frac{w \in M_D,\ (p[v \mapsto v'], \sigma[v' \mapsto w]) \xrightarrow{m}_t (p', \sigma')}{(\sum_{v:D} p, \sigma) \xrightarrow{m}_t (p', \sigma')}$$

$$(Sum^t)\frac{w \in M_D, (p, \sigma[v \mapsto w]) \rightsquigarrow_t}{(\sum_{v:D} p, \sigma) \rightsquigarrow_t}$$

where v' is a fresh variable of sort D, i.e., $v' \notin dom(\sigma)$

Table 2.2 SOS deduction rules for the sum operator

$$(Time^t_1)\frac{(p, \sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} \checkmark}{(p^c u, \sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} \checkmark} \qquad (Time^t_2)\frac{(p, \sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} (p', \sigma')}{(p^c u, \sigma) \xrightarrow{m}_{[\![u]\!]^\sigma} (p', \sigma')}$$

$$(Time^t)\frac{(p, \sigma) \rightsquigarrow_t}{(p^c u, \sigma) \rightsquigarrow_t}\ t < [\![u]\!]^\sigma$$

$$(Init^t_1)\frac{(p, \sigma) \xrightarrow{m}_t \checkmark}{(u \gg p, \sigma) \xrightarrow{m}_t \checkmark}\ [\![u]\!]^\sigma < t \qquad (Init^t_2)\frac{(p, \sigma) \xrightarrow{m}_t (p', \sigma')}{(u \gg p, \sigma) \xrightarrow{m}_t (p', \sigma')}\ [\![u]\!]^\sigma < t$$

$$(Init^{t\,\prime}_1)\frac{(p, \sigma) \rightsquigarrow_t}{(u \gg p, \sigma) \rightsquigarrow_t} \qquad (Init^{t\,\prime}_2)\frac{}{(u \gg p, \sigma) \rightsquigarrow_t}\ t < [\![u]\!]^\sigma$$

Table 2.3 SOS deduction rules for the time and the initialization operator

- E_i and E_j are data sorts over which the variables e_i and e_j range.
- $c_i : D \times E_i \to \mathbb{B}$ and $c_j : D \times E_j \to \mathbb{B}$ are Boolean functions. Hence, $c_i(d, e_i)$ and $c_j(d, e_j)$ are terms of sort $\mathbb{B}$ (denoting the set of Boolean values) that serve as a Boolean guard to allow actions.
- $\alpha_i(d, e_i)$ is a multi-action $a^1_i(f^1_i(d, e_i))| \cdots |a^{n_i}_i(f^{n_i}_i(d, e_i))$, where $f^k_i(d, e_i)$ (for $1 \leq k \leq n_i$) specifies the data parameters for the labeled action a^k_i.
- $\alpha_{\delta j}(d, e_j)$ is either δ or a multi-action $a^1_j(f^1_j(d, e_j))| \cdots |a^{n_j}_j(f^{n_j}_j(d, e_j))$, where $f^k_j(d, e_j)$ (for $1 \leq k \leq n_j$) gives the parameters of action name a^k_j.
- $t_i : D \times E_i \to \mathbb{R}$ and $t_j : D \times E_j \to \mathbb{R}$ respectively are the time stamp functions of multi-actions $\alpha_i(d, e_i)$ and $\alpha_{\delta j}(d, e_j)$.

$$(Par_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg q,\sigma)} \qquad (Par_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(p\|q,\sigma) \xrightarrow{m}_t (p'\|t \gg q,\sigma')}$$

$$(Par_3)\frac{(q,\sigma) \xrightarrow{m}_t \checkmark}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg p,\sigma)} \qquad (Par_4)\frac{(q,\sigma) \xrightarrow{m}_t (q',\sigma')}{(p\|q,\sigma) \xrightarrow{m}_t (t \gg p\|q',\sigma')}$$

$$(Par_5)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p\|q,\sigma) \xrightarrow{(m|n)_\sim}_t \checkmark} \qquad (Par_6)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark, (q,\sigma) \xrightarrow{n}_t (q',\sigma')}{(p\|q,\sigma) \xrightarrow{(m|n)_\sim}_t (q',\sigma')}$$

$$(Par_7)\frac{\begin{array}{c}(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\\ (q,\sigma) \xrightarrow{n}_t \checkmark\end{array}}{(p\|q,\sigma) \xrightarrow{(m|n)_\sim}_t (p',\sigma')} \qquad (Par_8)\frac{\begin{array}{c}(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\\ (q,\sigma) \xrightarrow{n}_t (q',\sigma'')\end{array}}{(p\|q,\sigma) \xrightarrow{(m|n)_\sim}_t (p'\|q',\sigma' \cup \sigma'')}$$

$$(Par^t)\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p\|q,\sigma) \rightsquigarrow_t}$$

Table 2.4 SOS deduction rules for the parallel operator

- $g_i : D \times E_i \rightarrow D$ denotes the next state function. Hence $g_i(d, e_i)$ is a term of sort D that denotes the next state.

2.4 Modal μ-Calculus

A behavioral *requirement* is a functional aspect of a behavioral specification. These are formulated positively (e.g., the system must perform an action a) or are formulated negatively (e.g., the system may never perform an action a). To define requirements for mCRL2 specifications, we use a logic that is based on [HM80], namely the modal μ-calculus [EC80, Koz83]. The modal μ-calculus that the mCRL2 language uses is an extension that supports data, regular expressions and time.

Modal μ-calculus formulas are verified against a behavioral model as described by an mCRL2 specification. The requirements that are specified within this thesis use a restricted fragment of the modal μ-calculus. The restricted fragment is described by the following grammar:

$$\begin{array}{ll} \alpha ::= & a(\overrightarrow{d}) \mid \overline{\alpha} \mid \alpha|\alpha \mid true \\ \rho ::= & \alpha \mid \rho \cdot \rho \mid \rho^\star \\ \phi ::= & false \mid true \mid \neg\phi \mid \phi \Rightarrow \phi \mid \phi \wedge \phi \mid [\rho]\phi \mid \langle\rho\rangle\phi \mid \forall_{x:D}\phi \end{array}$$

Here, α represents a set of multi-actions, ρ represents a sequences of multi-actions, and ϕ represents a property.

$$(Lmerge_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(p \lfloor\!\lfloor q,\sigma) \xrightarrow{m}_t (t \gg q,\sigma)} \qquad (Lmerge_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(p \lfloor\!\lfloor q,\sigma) \xrightarrow{m}_t (p' \| t \gg q,\sigma')}$$

$$(Lmerge^t)\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p \lfloor\!\lfloor q,\sigma) \rightsquigarrow_t}$$

$$(Sync_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p|q,\sigma) \xrightarrow{(m|n)\sim}_t \checkmark} \qquad (Sync_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \xrightarrow{n}_t \checkmark}{(p|q,\sigma) \xrightarrow{(m|n)\sim}_t (p',\sigma')}$$

$$(Sync_3)\frac{\begin{array}{c}(p,\sigma) \xrightarrow{m}_t \checkmark,\\ (q,\sigma) \xrightarrow{n}_t (q',\sigma')\end{array}}{(p|q,\sigma) \xrightarrow{(m|n)\sim}_t (q',\sigma')} \qquad (Sync_4)\frac{\begin{array}{c}(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\\ (q,\sigma) \xrightarrow{n}_t (q',\sigma'')\end{array}}{(p|q,\sigma) \xrightarrow{(m|n)\sim}_t (p' \| q',\sigma' \cup \sigma'')}$$

$$(Sync^t)\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p|q,\sigma) \rightsquigarrow_t}$$

$$(Before_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark,\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \xrightarrow{m}_t \checkmark} \qquad (Before_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma'),\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \xrightarrow{m}_t (p',\sigma')}$$

$$(Before^t)\frac{(p,\sigma) \rightsquigarrow_t,\ (q,\sigma) \rightsquigarrow_t}{(p \ll q,\sigma) \rightsquigarrow_t}$$

Table 2.5 SOS deduction rules for the auxiliary parallel operators

An action labeled a with a vector data parameters d is denoted as $a(\overrightarrow{d})$. The absence of an multi-action α corresponds to $\overline{\alpha}$. A multi-action constructed from two multi-actions is denoted by $\alpha|\alpha$. An arbitrary multi-action is denoted by *true*. The concatenation of two action sequences is described by the notation $\rho \cdot \rho$. To describe the iteration of an action sequence, i.e., the reflexive transitive closure of an action sequence, we use $\rho^\star$.

The property *true* holds in every state of a model and *false* if the property holds for no model. The property $\phi \Rightarrow \psi$ holds if the property ϕ and the property ψ hold. The property $\phi \wedge \psi$ states that both ϕ and ψ must hold. The property $[\rho]\phi$ states the property that ϕ holds in all states that can be reached by a sequence described by ρ. The property $\langle\rho\rangle\phi$ describes that ϕ holds in some state that can be reached by a sequence from ρ. A more elaborate description of the modal μ-calculus and its semantics can be found in [Bra92, GM99].

To verify a modal μ-calculus formula, one can first transform an mCRL2 specification into its linear case, i.e., into an LPS. Then the LPS, together with a modal μ-calculus formula can be encoded into a Parameterised Boolean Equation System (PBES) [GW05a, OW10]. The solution of the PBES then reflects the solution of the

$$(Allow_1)\frac{\underline{m} \cap (V \cup \{\tau\}) \neq \emptyset, \quad (p,\sigma) \xrightarrow{m}_t \checkmark}{(\nabla_V(p),\sigma) \xrightarrow{m}_t \checkmark} \quad (Allow_2)\frac{\underline{m} \cap (V \cup \{\tau\}) \neq \emptyset, \quad (p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\nabla_V(p),\sigma) \xrightarrow{m}_t (\nabla_V(p'),\sigma')} \quad (Allow^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\nabla_V(p),\sigma) \rightsquigarrow_t}$$

$$(Block_1)\frac{m^{\{\}} \cap B = \emptyset, \quad (p,\sigma) \xrightarrow{m}_t \checkmark}{(\partial_B(p),\sigma) \xrightarrow{m} \checkmark} \quad (Block_2)\frac{m^{\{\}} \cap B = \emptyset, \quad (p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\partial_B(p),\sigma) \xrightarrow{m}_t (\partial_B(p'),\sigma')} \quad (Block^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\partial_B(p),\sigma) \rightsquigarrow_t}$$

$$(Ren_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\rho_R(p),\sigma) \xrightarrow{R\bullet(m)}_t \checkmark} \quad (Ren_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\rho_R(p),\sigma) \xrightarrow{R\bullet(m)}_t (\rho_R(p'),\sigma')} \quad (Ren^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\rho_R(p),\sigma) \rightsquigarrow_t}$$

$$(Comm_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)}_t \checkmark} \quad (Comm_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\Gamma_C(p),\sigma) \xrightarrow{\gamma_C(m)}_t (\Gamma_C(p'),\sigma')} \quad (Comm^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\Gamma_C(p),\sigma) \rightsquigarrow_t}$$

$$(Hide_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)}_t \checkmark} \quad (Hide_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\tau_I(p),\sigma) \xrightarrow{\theta_I(m)}_t (\tau_I(p'),\sigma')} \quad (Hide^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\tau_I(p),\sigma) \rightsquigarrow_t}$$

$$(Pre_1)\frac{(p,\sigma) \xrightarrow{m}_t \checkmark}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)}_t \checkmark} \quad (Pre_2)\frac{(p,\sigma) \xrightarrow{m}_t (p',\sigma')}{(\Upsilon_U(p),\sigma) \xrightarrow{\eta_U(m)}_t (\Upsilon_U(p'),\sigma')} \quad (Pre^t)\frac{(p,\sigma) \rightsquigarrow_t}{(\Upsilon_U(p),\sigma) \rightsquigarrow_t}$$

Table 2.6 SOS deduction rules for the auxiliary operators

$$(Def_1)\frac{(q,\sigma[\overrightarrow{v} \mapsto \{[\overrightarrow{d}]\}^\sigma]) \xrightarrow{m}_t \checkmark}{(X(\overrightarrow{v}=\overrightarrow{d}),\sigma) \xrightarrow{m}_t \checkmark} \quad (Def_2)\frac{(q[\overrightarrow{v} \mapsto \overrightarrow{v'}],\sigma[\overrightarrow{v'} \mapsto \{[\overrightarrow{d}]\}^\sigma]) \xrightarrow{m}_t (q',\sigma')}{(X(\overrightarrow{v}=\overrightarrow{d}),\sigma) \xrightarrow{m}_t (q',\sigma')}$$

$$(Def^t)\frac{(q,\sigma[\overrightarrow{v} \mapsto \{[\overrightarrow{d}]\}^\sigma]) \rightsquigarrow_t}{(X(\overrightarrow{v} = \overrightarrow{d}),\sigma) \rightsquigarrow_t}$$

where $X(\overrightarrow{v}:\overrightarrow{D}) = q \in PE$ and $\overrightarrow{v'}$ are fresh variables of sort $\overrightarrow{D}$ with respect to σ, i.e., $\overrightarrow{v'} \notin dom(\sigma)$

Table 2.7 SOS deduction rules for recursion

model-checking problem. A way in which PBESs are solved is described and illustrated in [GW05b, PWW11]. The encoding can be performed fully automatically for the first-order modal μ-calculus and infinite-state models.

Part I

Syntactically Engineered Models

Chapter 3

Modeling System Descriptions

3.1 Introduction

Creating a system that meets the behavioral requirements that have been agreed upon before development is a challenging task. To predict that a system complies, engineers can create formal models that are subjected to validation and verification techniques. As every specification language has its own characteristics, it is very important to select a suitable language to model the system's behavior. This chapter explores the modeling of system behavior based on a set of informal descriptions of a concurrent system. It describes an instance of a bridge between the specification environment and the analysis environment from Figure 1.1.

There are different ways to specify the behavior of a system. Many formal specification languages seem suitable for describing the system's behavior, when they are applied in case studies or toy examples. In many cases, these systems are specially selected or tailored to assess certain features in a language. Unfortunately, when actual systems are modeled, it often turns out that a specification language is unable to express all of the envisioned behavior, because the semantics is not vigorous enough to express the complex behavioral patterns. In these cases, engineers are required to deviate from the system's behavior, have to apply abstractions such that the inexpressible behavior is removed, need to select a different specification language, or can simply not verify all desired properties.

To guide (modeling) engineers in selecting a suitable language, the authors of [DS09] compare the four specification languages TLA+ [Lam02], Bluespec [HA00], Statecharts [Har87], and ACP [BK84] in a selected case study. They compare the languages w.r.t. the following three criteria:

1. the amount of *local* reasoning (as opposed to global and temporal) that is required by the engineer to specify behavior,

2. the *adaptability* that is required to make variations in design intent, and
3. the ability of a language to *capture* the intended design.

To evaluate the different criteria, they specify a switch that internally routes packets between buffers. The routing is described by a set of rules that (i) specify priority among packets and (ii) allow for a simultaneous packet transfer when it complies to certain criteria. As these rules turn out to be complementary, they illustrate contradictory concerns which emphasize the possible weaknesses in the different languages. They conclude that each of the specification languages performs poorly in at least two of these criteria.

In extension to the case study presented in [DS09], this chapter evaluates the mCRL2 language [GMR+06] on the same criteria. Thereby we show that the mCRL2 specification language is better suited than the other specification languages, with respect to the presented case study. In addition, we also verify a couple of requirements with the help of the mCRL2 toolset, to validate that the intended behavior is actually modeled.

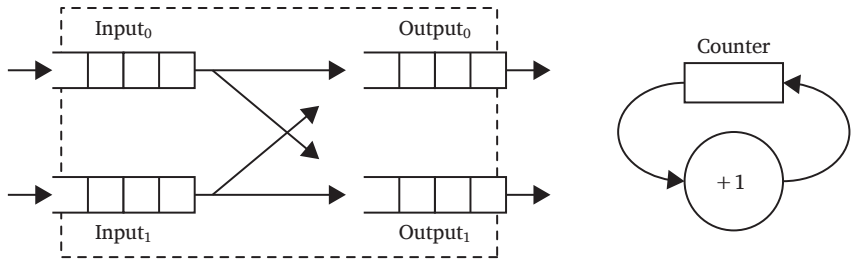
The models are constructed in a relatively straightforward way from the informal description. It turns out that the required multi-party communication is captured by the advanced communication mechanisms of the mCRL2 language. Although the mCRL2 language has no direct notions to specify priority, it is possible to express the different priority types for the cases at hand.

This chapter is structured as follows. The description of the switches and the ways in which they are modeled are explained in Chapters 3.2, 3.3 and 3.4. Chapter 3.5 elaborates on the verified requirements. Chapter 3.6 compares our work to that of others. Chapter 3.7 describes our conclusions and future work.

3.2 Specification of the Simplified 2×2 Switch

The original specification of the 2×2 switch is first mentioned in [Blu05]. The case study that is discussed in this chapter, describes the original specification and two variations. These specifications are referred to as the “Original Switch”, the “Simplified Switch”, and the “Modified Switch”, respectively. In the specification we follow the informal description from [DS09] as closely as possible. This means that we introduce a single process for each of the four buffers. By means of the advanced communication mechanisms offered in the mCRL2 language, we describe their non-trivial interaction. In this and in the following two sections, we discuss the modeling of the different specifications and explain how we have resolved the posed challenges.

The Simplified Switch consists of two input FIFO buffers and two output FIFO buffers. All buffers have a unique identifier, w.r.t. the type of buffer. That is, the identifier of an input or output buffer corresponds to either the numerical values 0 or 1. Furthermore we assume that all buffers have the same capacity. So, we assign to all of them the same (finite amount of) capacity for storing packets. Figure 3.1(a) illustrates the Simplified Switch.



(a) A 2×2 switch (b) A packet counter

Figure 3.1 A 2×2 switch and a counter

Every packet consists of 32 bits. Packets enter the system via the input buffers and depart the system via the output buffers. Packets are transferred from an input buffer to one of the output buffers based on the first bit of a packet: When a first bit is 0, it is routed to the output buffer with identity 0. Otherwise, the packet is routed to the output buffer with identity 1.

The packets may only be transferred when the relevant output buffer is not full. A buffer operates per clock cycle and performs at most one operation, namely the receive a packet, the send a packet, or do nothing. Furthermore, we require maximal throughput, i.e., a packet is transferred if it has the ability to do so. When packets from different input buffers want to transfer to the same output buffer, the transfer of the packet from input buffer 0 gets priority over the transfer of the packet from input buffer 1.

3.2.1 Bits and Packets

The data type of bits consists of two different values. In an mCRL2 specification, this is modeled as:

sort $\quad Bit = \textbf{struct}\ zero \mid one;$

In the case study packets consist of 32 bits. This implies that a single packet is represented by 2^{32} different configurations. The mCRL2 language allows the description of such a data type without any problems. A specification that models a packet is represented by a structured sort that composes the 32 bits by:

sort $\quad Packet = \textbf{struct}\ packet(b_1, b_2, \ldots, b_{32} : Bit);$

From a modeling point of view, we do not object to such a representation or see any difficulty to write it down in an mCRL2 specification. Unfortunately, for a formal analysis with tools that generate explicit state spaces for verification, this has an

apparent drawback. The specification above gives rise to 2^{32} different potential contents for each position in each of the considered buffers. This number is usually too big to be handled by current state-of-the-art model-checking tools. For that reason we introduce an appropriate abstraction.

From the description of the Simplified Switch, we deduce that only the first bit of a packet is relevant. According to the first bit, packets are routed to output buffer 0 if the first bit of the packet is 0. In all other cases the packets are routed to output buffer 1. Hence, we abstract from the irrelevant bits of a packet and only model the first bit. Consequently, the structure of a packet is redefined as:

sort $\mathit{Packet} = \textbf{struct}\ \mathit{packet}(b_1 : \mathit{Bit});$

Next we introduce a function that routes packets to their proper destinations. So, we define a mapping *dest* that expresses the relation between the data of a packet and the destination output buffer.

map $\mathit{dest} : \mathit{Packet} \rightarrow \mathbb{N};$
eqn $\mathit{dest}(\mathit{packet}(\mathit{zero})) = 0;$
$\mathit{dest}(\mathit{packet}(\mathit{one})) = 1;$

3.2.2 Capacity of the Buffers

The system consists of four buffers, for which each buffer is modeled as a list of packets. Each buffer has the same capacity *cap*. It is assumed to be at least 1. To specify the size of the buffers in the specifications, without referring to an explicitly defined value we introduce a constant that models the size of the buffers.

map $\mathit{cap} : \mathbb{N}^+;$

By means of an equation we assign a specific value to this mapping. The restriction is only added to limit the size of the generated state space during the analysis. When desired, the capacity for all the buffers can be changed in one place.

eqn $\mathit{cap} = 3;$

3.2.3 Information Exchange between Processes

To observe that packets enter and leave the 2×2 switch, two actions with data parameters are introduced. One action adds a packet to an input buffer (*enter*). The other action removes a packet from an output buffer (*leave*). Both actions carry two data parameters. The first data parameter refers to the identity of an input buffer (for *enter*-actions) or an output buffer (in case of *leave*-actions). The second data parameter represents the actual (abstracted) data from a packet.

act $\mathit{enter} : \mathbb{N} \times \mathit{Packet};$
$\mathit{leave} : \mathbb{N} \times \mathit{Packet};$

Sending a packet from an input buffer to an output buffer is described by the *send* action. Similarly, the receipt of a packet by an output buffer is described by the action *recv*. To synchronize actions, the mCRL2 language provides synchronous communication between processes, when all the data parameters in the synchronizing actions have the same value. To reflect a successful synchronization of a *send* and a *recv*action, we use the action *comm*.

The actions *send*, *recv* and *comm* are modeled with three data parameters. The first parameter denotes the identity of the input buffer that sends a packet. The second parameter denotes the identity of the output buffer that receives a packet. The third parameter denotes the data packet that is transferred between the buffers. The first and second parameter provide handles to observe the routing of packets, i.e., they are used to express and verify requirements in Chapter 3.5. The last data parameter is required to transfer and observe the data flow between the buffers. Note that the second parameter is a cosmetic addition that could have been derived from the data of a packet.

$$\begin{array}{lll} \textbf{act} & \mathit{send} & : \mathbb{N} \times \mathbb{N} \times \mathit{Packet}; \\ & \mathit{recv} & : \mathbb{N} \times \mathbb{N} \times \mathit{Packet}; \\ & \mathit{comm} & : \mathbb{N} \times \mathbb{N} \times \mathit{Packet}; \end{array}$$

The packet exchange between an input buffer and an output buffer depends on the content of the other input buffer. In the mCRL2 language it is possible to use multi-party communication to establish the involvement of another process. This means that we require actions that reveal information about a third party in the communication. We introduce the actions *grant* and *free* for this purpose. Both $\mathit{grant}(i, o, p)$ and $\mathit{free}(i, o, p)$ denote that input buffer i is granted permission to send a packet p to output buffer o. The first action establishes priority among packets. The second action enables in the simultaneous packet transfer. A more detailed explanation is provided later on in this section.

$$\begin{array}{lll} \textbf{act} & \mathit{grant} & : \mathbb{N} \times \mathbb{N} \times \mathit{Packet}; \\ & \mathit{free} & : \mathbb{N} \times \mathbb{N} \times \mathit{Packet}; \end{array}$$

3.2.4 Output Buffers with Capacity *cap*

In the mCRL2 language, a FIFO buffer *Output* with capacity *cap* is modeled by the following process specification:

$$\begin{array}{ll} \textbf{proc} & \mathit{Output}(i : \mathbb{N}, c : \mathit{List}(\mathit{Packet})) = \\ & \quad \#c < \mathit{cap} \rightarrow \sum_{s:\mathbb{N}} \sum_{p:\mathit{Packet}} \mathit{recv}(s, i, p) \cdot \mathit{Output}(i, p \triangleright c) \\ & + \quad c \not\approx [\,] \quad \rightarrow \mathit{leave}(i, \mathit{rhead}(c)) \cdot \mathit{Output}(i, \mathit{rtail}(c)); \end{array}$$

The first line specifies the name of the process and declares the associated process parameters. The buffer process has two parameters. The first process parameter represents the identity of an output buffer. The second process parameter captures the

contents of the buffer as a list of packets. As already described, a buffer receives arbitrary packets as long as the buffer is not yet full ($\#c$ denotes the number of elements in the list c). With this guard we model the first summand. We specify that a received packet is appended to the buffer. Appending a packet p to a buffer contents c is denoted by $p \triangleright c$. The second summand describes the sending from, and the removal of the first packet in a buffer. Therefore we ensure that the buffer is not empty ($c \not\approx []$) before we perform send action (*leave*) of the first packet ($rhead(c)$) and remove the packet from the buffer ($rtail(c)$). By modeling a buffer like this, the specification of the output buffer does not rely on the acceptance of packets with a specific first bit, i.e., it accepts packets regardless of their content. The output buffer performs at most one action at a time.

3.2.5 Input Buffers with Capacity *cap*

The main challenges of this modeling exercise are (i) to deal appropriately with the priority of input buffer 0 over input buffer 1 in case both buffers want to transfer a packet to the same destination, and (ii) to deal with the required simultaneous packet transfer when both buffers need to transfer packets to different destinations. In this section we gradually shape the mCRL2 specification by first specifying the input buffers, and then defining the interaction between the different processes.

We model the behavior of an input buffer analogously to that of an output buffer:

$$
\begin{aligned}
\textbf{proc}\quad & Input(i:\mathbb{N}, c:List(Packet)) = \\
& \quad \#c < cap \rightarrow \sum_{p:Packet} enter(i,p) \cdot Input(i, p \triangleright c) \\
& + \quad c \not\approx [] \quad \rightarrow send(i, dest(rhead(c)), rhead(c)) \cdot Input(i, rtail(c));
\end{aligned}
$$

Next, we setup the basic communication between input and output buffers. We first specify that the four buffers run in parallel. Furthermore, we specify that a successful synchronization of *send* and *recv* actions, result in *comm* actions. This is expressed by the subscript parameter $send \mid recv \rightarrow comm$ in the communication operator Γ. We only allow successful communications. Therefore we encapsulate all *send* and *recv* actions that do not result in a successful synchronization. In this way, the insertion and the removal of a packet can be performed simultaneously by different buffers, while among the remaining buffers packets transfer are enabled. By combining the instantiated process definitions with the communication and encapsulation operators, we obtain the following initialization:

$$
\begin{aligned}
\textbf{init}\quad & \partial_{\{send,recv\}}(\Gamma_{\{send|recv\rightarrow comm\}}(\\
& \quad Input(0,[]) \parallel Input(1,[]) \parallel Output(0,[]) \parallel Output(1,[])));
\end{aligned}
$$

To acquire the simultaneous packet transfer and prioritized packet transfer, the specification is adapted in two ways. The first step models the prioritized packet transfer when packets route to the same destination. The second step models the required simultaneous packet transfer to different output buffers.

Prioritized packet transfer When packets are transferred to the same output buffer, the input buffer with the lowest identifier has priority over the other sending input buffer. The way in which we model a prioritized packet transfer is as follows. The input buffer signals the transfers that are allowed for execution by the other input buffer by means of the *grant*-action. If a buffer is empty it grants permission for any transfer in the other process of the input buffer. If the buffer is not empty it only grants permission to packet transfers that originate from input buffers that have a lower identity.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Input}(i:\mathbb{N}, c:\mathit{List}(\mathit{Packet})) = \\
& \quad \#c < \mathit{cap} \rightarrow \sum_{p:\mathit{Packet}} \mathit{enter}(i,p) \cdot \mathit{Input}(i, p \triangleright c) \\
& + \quad c \not\approx [] \quad \rightarrow \mathit{send}(i, \mathit{dest}(\mathit{rhead}(c)), \mathit{rhead}(c)) \cdot \mathit{Input}(i, \mathit{rtail}(c)); \\
& + \quad c \approx [] \quad \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\mathit{Packet}} \mathit{grant}(n,m,p) \cdot \mathit{Input}(i,c) \\
& + \quad c \not\approx [] \quad \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow \mathit{grant}(n, \mathit{dest}(\mathit{rhead}(c)), \mathit{rhead}(c)) \cdot \mathit{Input}(i,c)
\end{array}
$$

To ensure that *grant*-actions synchronize with corresponding *send*- and *recv*-actions, a second communication function is added:

$$
\begin{array}{ll}
\textbf{init} & \partial_{\{\mathit{send},\mathit{recv},\mathit{grant}\}}(\Gamma_{\{\mathit{send}|\mathit{recv}\rightarrow\mathit{comm}\}}(\Gamma_{\{\mathit{send}|\mathit{recv}|\mathit{grant}\rightarrow\mathit{comm}\}}(\\
& \quad \mathit{Input}(0,[]) \parallel \mathit{Input}(1,[]) \parallel \mathit{Output}(0,[]) \parallel \mathit{Output}(1,[]))));
\end{array}
$$

By nesting the communications we ensure that the innermost communication has the highest priority. Because the communication is applied first to multi-action *send* | *recv* | *grant*, instead of multi-action *send* | *recv*, we know that the priority is either granted by (i) a buffer having a higher identifier that wants to send a packet to the same output buffer or (ii) the other input buffer is empty. Because *send* | *recv* only communicates when both (i) and (ii) do not apply, we know that packets resulting from the second communication are routed to different destinations.

Maximal communication To enforce that packets are transferred simultaneously, we introduce an announcement. That is, if a packet is routed to a destination, it announces to the other input buffer that a simultaneous transfer is enabled for packets routed to another destination. The announcement is modeled by means of the *free*-action.

$$
\begin{array}{ll}
\textbf{proc} & \mathit{Input}(i:\mathbb{N}, c:\mathit{List}(\mathit{Packet})) = \\
& \quad \#c < \mathit{cap} \rightarrow \sum_{p:\mathit{Packet}} \mathit{enter}(i,p) \cdot \mathit{Input}(i, p \triangleright c) \\
& + \quad c \not\approx [\,] \quad \rightarrow \sum_{n:\mathbb{N}} \sum_{p:\mathit{Packet}} n \not\approx i \wedge \mathit{dest}(p) \not\approx \mathit{dest}(\mathit{rhead}(c)) \rightarrow \\
& \qquad\qquad \mathit{send}(i, \mathit{dest}(\mathit{rhead}(c)), \mathit{rhead}(c)) \mid \mathit{free}(n, \mathit{dest}(p), p) \\
& \qquad\qquad \cdot \mathit{Input}(i, \mathit{rtail}(c)) \\
& + \quad c \not\approx [\,] \quad \rightarrow \mathit{send}(i, \mathit{dest}(\mathit{rhead}(c)), \mathit{rhead}(c)) \cdot \mathit{Input}(i, \mathit{rtail}(c)) \\
& + \quad c \approx [\,] \quad \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:\mathit{Packet}} \mathit{grant}(n,m,p) \cdot \mathit{Input}(i,c) \\
& + \quad c \not\approx [\,] \quad \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow \mathit{grant}(n, \mathit{dest}(\mathit{rhead}(c)), \mathit{rhead}(c)) \cdot \mathit{Input}(i,c);
\end{array}
$$

By changing the communication function to $\{\mathit{send} \mid \mathit{recv} \mid \mathit{free} \rightarrow \mathit{comm}\}$ we only allow packet transfers when the other input buffer grants permission and that a transfer to different destinations is performed simultaneously. This way simultaneous packet transfers are achieved whenever possible. All possible communications are now permitted by either a *grant*- or a *free*-action.

$$
\begin{array}{ll}
\textbf{init} & \partial_{\{\mathit{send},\mathit{recv},\mathit{grant},\mathit{free}\}}(\\
& \quad \Gamma_{\{\mathit{send}\mid\mathit{recv}\mid\mathit{free}\rightarrow\mathit{comm}\}}(\Gamma_{\{\mathit{send}\mid\mathit{recv}\mid\mathit{grant}\rightarrow\mathit{comm}\}}(\\
& \qquad \mathit{Input}(0,[\,]) \parallel \mathit{Input}(1,[\,]) \parallel \mathit{Output}(0,[\,]) \parallel \mathit{Output}(1,[\,]))));
\end{array}
$$

The order in which the communications are applied is now of no importance. Specifying the two communications as single one is not allowed, since the left-hand sides of the communication patterns share action labels, which might lead to a non-unique solution. For this reason we distribute the communication functions over the two separate communications. To provide (partial) evidence that the order is unimportant, we have used the mCRL2 toolset. Here we generate the respective LTSs for the buffers of capacity 1, 2 and 3, and show that the corresponding state spaces are strongly bisimilar (even isomorphic). The tools that have been used are:

1. mcrl22lps turns an mCRL2 specification as an LPS.
2. lps2lts exhaustively explores an LPS and stores the exploration result in an LTS.
3. ltscompare determines whether two LTSs are related by some equivalence or preorder. In these examples we have used the strong bisimilarity equivalence relation.

3.3 Specification of the Original 2×2 Switch

The Original Switch is an extension of the Simplified Switch. The Original switch contains an additional counter, that counts interesting packets that are transferred between input and output buffers. A packet is considered interesting if its second, third, and fourth bit are all 0. The counter is restricted, such that the value can only

be incremented once every clock cycle. So when both input buffers are capable of transferring interesting packets, priority is given to the transfer from input buffer 0 and the transfer from input buffer 1 is delayed. Hence, we now may only transfer packets simultaneously, if either one of them is not interesting. Otherwise a process needs to either take or grant priority like in the Simplified Switch specification. Figure 3.1(b) illustrates the counter. Figure 3.1 depicts both the Simplified Switch and the Modified Switch.

This section adapts the model of the Simplified Switch to a model that corresponds to the design intent of the Original Switch. Thereto, we extend a part of the data specification and adapt the behaviors of the buffer processes slightly.

3.3.1 Packets

The fact that the second, third and fourth bit of a packet have become relevant for the behavior implies that we need to reconsider our definition of the data type that represents a packet. We represent a packet as four bits (i.e., the relevant ones) in a way similar to the current definition. Instead, and more abstractly, we decide to model packets as before but with an additional Boolean parameter that indicates if a packet is interesting (*true*) or not (*false*).

sort $\mathit{Packet} = \textbf{struct}\ \mathit{packet}(b_1 : \mathit{Bit},\ \mathit{int} : \mathbb{B});$

By extending the structured sort, we need to update the destination function for routing packets as well. As the second, third, and fourth bit have no effect on the routed destination, the adaptation is straightforward.

map $\mathit{dest} : \mathit{Packet} \rightarrow \mathbb{N};$
var $b : \mathbb{B};$
eqn $\mathit{dest}(\mathit{packet}(\mathit{zero}, b)) = 0;$
$\mathit{dest}(\mathit{packet}(\mathit{one}, b)) = 1;$

3.3.2 The Act of Counting

There are several ways to model the act of counting interesting packets. One way is to introduce an action data parameter that reflects the amount of interesting packets that have been transferred. Another way is to introduce an action that indicates that such a packet is transferred. We have chosen the second solution, since it creates a finite LTS if we generate the explicit transitions. Thus, the counting of interesting packets is reflected by performing the action *inc* without any data parameters.

act *inc*;

Another decision that needs to be made is which entity performs the counting. One solution is to introduce a separate process. Another option is to extend the functionality for either the input or the output buffers. We choose to extend the

functionality of the output buffers, since the modifications are performed in a local processes opposed to creating a process that acts as a governing controller. Note, that implementing the other solution poses no real problems for the mCRL2 language.

To accommodate this behavior, the first summand of the output buffer from the Simplified Switch is split into two cases, one for receiving and counting interesting packets and one for receiving non-interesting packets. To decide if a packet is interesting, the projection function *int* is used. The projection function for a specific field of a structured sort is specified in the sort declaration. For the sort *Packet* we use the projection functions b_1 and *int* for obtaining the values of the first and second field, respectively.

$$
\begin{aligned}
\textbf{proc}\quad & Output(i:\mathbb{N}, c:List(Packet)) = \\
& \quad \#c < cap \rightarrow \sum_{s:\mathbb{N}} \sum_{p:Packet} \big(\quad int(p) \ \rightarrow recv(s,i,p) \mid inc \cdot Output(i, p \triangleright c) \\
& \qquad\qquad\qquad\qquad\qquad\qquad + \neg int(p) \rightarrow recv(s,i,p) \cdot Output(i, p \triangleright c)\big) \\
& + \quad c \not\approx [] \qquad \rightarrow leave(i, rhead(c)) \cdot Output(i, rtail(c));
\end{aligned}
$$

3.3.3 Adapting the Input Buffer

The Original Switch poses an additional restriction on the communication between the input and output buffer. We only transfer packets simultaneously if they have different destinations and at most one packet is interesting. This is expressed in the second summand. When both input buffers contain interesting packets and these packets need to be routed to different destinations, priority is granted to any input buffer with a lower identity. This is described by the fifth summand below. Furthermore we must grant priority to both interesting and non-interesting packets when local packets are non-interesting. For that reason we adapt the last summand.

$$
\begin{aligned}
\textbf{proc}\quad & Input(i:\mathbb{N}, c:List(Packet)) = \\
& \#c < cap \ \rightarrow \sum_{p:Packet} enter(i,p) \cdot Input(i, p \triangleright c) \\
& + c \not\approx [] \ \rightarrow \sum_{p:Packet} dest(p) \not\approx dest(rhead(c)) \wedge (\neg int(p) \vee \neg int(rhead(c))) \\
& \qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n \not\approx i \ \rightarrow send(i, dest(rhead(c)), rhead(c)) \mid free(n, dest(p), p) \\
& \qquad\qquad\qquad\qquad\qquad\qquad \cdot \ Input(i, rtail(c)) \\
& + c \not\approx [] \ \rightarrow send(i, dest(rhead(c)), rhead(c)) \cdot Input(i, rtail(c)) \\
& + c \approx [] \ \rightarrow \sum_{n,m:\mathbb{N}} \sum_{p:Packet} grant(n,m,p) \cdot Input(i,c) \\
& + c \not\approx [] \wedge \ int(rhead(c)) \rightarrow \sum_{p:Packet} dest(p) \approx dest(rhead(c)) \vee int(p) \\
& \qquad\qquad\qquad\qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow grant(n, dest(p), p) \cdot Input(i,c) \\
& + c \not\approx [] \wedge \neg int(rhead(c)) \rightarrow \sum_{p:Packet} b_1(p) \approx b_1(rhead(c)) \\
& \qquad\qquad\qquad\qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n < i \ \rightarrow grant(n, dest(rhead(c)), p) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \cdot \ Input(i,c);
\end{aligned}
$$

3.4 Specification of the Modified 2×2 Switch

The modified 2×2 switch is derived from the original 2×2 switch. The modified version alters the way in which the priority is handled for colliding packets that are both interesting but have different routing destinations. The modifications are described by two predicates, namely "both packet transfers have the same destination" (C1) and "both packet transfers are interesting" (C2). If either one of these predicates holds, priority is assigned to a packet transfer that originates from input buffer 0. When C1 holds the first input buffer (input buffer 0) has priority over the second input buffer (input buffer 1). However if C1 does not hold and C2 holds, priority is assigned to a packet transfer that originates from the second input buffer.

To incorporate these predicates, we only have to alter the model of the input buffers. For the relevant cases, priority is granted to the input buffer with a higher identity. The relevant cases only affect the penultimate summand of the input buffer of the specification from the Original Switch. Hence, we take this summand and split it into two separate summands that respectively model the behavior of the predicates C1 and C2:

$$
\begin{aligned}
\textbf{proc}\quad & Input(i:\mathbb{N}, c:List(Packet)) = \\
& \#c < cap \;\rightarrow\; \sum_{p:Packet} enter(i,p) \cdot Input(i, p \triangleright c) \\
& +\; c \not\approx [] \;\rightarrow\; \sum_{p:Packet} dest(p) \not\approx dest(rhead(c)) \wedge (\neg int(p) \vee \neg int(rhead(c))) \\
& \qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n \not\approx i \rightarrow send(i, dest(rhead(c)), rhead(c)) \mid free(n, dest(p), p) \\
& \qquad\qquad \cdot\; Input(i, rtail(c)) \\
& +\; c \not\approx [] \;\rightarrow send(i, dest(rhead(c)), rhead(c)) \cdot Input(i, rtail(c)) \\
& +\; c \approx [] \;\rightarrow\; \sum_{n,m:\mathbb{N}} \sum_{p:Packet} grant(n,m,p) \cdot Input(i,c) \\
& + c \not\approx [] \wedge int(rhead(c)) \;\rightarrow\; \sum_{p:Packet} dest(p) \approx dest(rhead(c)) \\
& \qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n < i \rightarrow grant(n, dest(p), p) \cdot Input(i,c) \\
& + c \not\approx [] \wedge int(rhead(c)) \;\rightarrow\; \sum_{p:Packet} dest(p) \not\approx dest(rhead(c)) \wedge int(p) \\
& \qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n > i \rightarrow grant(n, dest(p), p) \cdot Input(i,c) \\
& + c \not\approx [] \wedge \neg int(rhead(c)) \rightarrow \sum_{p:Packet} b_1(p) \approx b_1(rhead(c)) \\
& \qquad\qquad \rightarrow \sum_{n:\mathbb{N}} n < i \;\rightarrow grant(n, dest(rhead(c)), p) \\
& \qquad\qquad\qquad\qquad \cdot\; Input(i,c);
\end{aligned}
$$

3.5 Properties of the Models

In [DS09], the authors presented their models without any form of formal verification. According to their descriptions of the models written in the Statechart formalism, the authors had to conclude that their specifications for the Original Switch did

not meet the design intent. To illustrate, their model contained a flaw when both buffers had interesting head packets and one of the buffers was full while the other was not. In that case, one packet should be delayed while the other head packet was routed. This however was not covered.

To prevent making errors similar to [DS09], and to convince readers that our specifications capture the correct design intent, we formulate a couple of requirements and verify that the provided models satisfy them. These requirements relate to deadlock analysis (Chapter 3.5.1), overflowing buffers (Chapter 3.5.2), packet collision (Chapter 3.5.3) and maximal progress (Chapter 3.5.4). The requirements are expressed in modal μ-calculus formulae (Chapter 2.4). The formulas are verified with the help of the mCRL2 toolset. The results are discussed in Chapter 3.5.5.

3.5.1 Deadlock Detection

Deadlock is a specific condition that brings the system into a halt, from which it cannot execute any behavior for any future. Deadlock can be caused by various reasons, among others due to circular resource dependencies or when all concurrent processes cannot fulfill their preconditions to execute any action.

We claim that all of the presented models are free from deadlock. Deadlock freedom is expressed by the following modal μ-calculus formula:

$$[true^{\star}]\langle true\rangle true \tag{1}$$

Informally, the above formula states that it must hold that for all possible paths it is possible to perform a next action.

3.5.2 Absence of Overflowing Buffers

We have used the standard mCRL2 container sort for the construction of lists to model the content of a buffer. Though, as the lengths of these lists are not fixed or bounded from above, the use in combination with the constant *cap* should guarantee that the buffers cannot overflow. To verify that the buffers indeed stay between these bounds we extend the model. Therefore we add the alternative summands to the input buffers:

$$\#c > cap \rightarrow overflow \cdot Input(i, c)$$

and add the summands to the output buffers:

$$\#c > cap \rightarrow overflow \cdot Output(i, c)$$

The extension has no impact on the routing of packets, because performing such an action does not affect the state of a model. The action is a witness when the model resides in an illegal state. To verify that the situation never occurs we state the

following modal formula:

$$
\begin{aligned}
& [true^{\star} \cdot overflow]false \\
\wedge\ & [true^{\star} \cdot overflow \mid overflow]false \\
\wedge\ & [true^{\star} \cdot overflow \mid overflow \mid overflow]false \\
\wedge\ & [true^{\star} \cdot overflow \mid overflow \mid overflow \mid overflow]false
\end{aligned} \tag{2}
$$

The formula states that there is no reachable state in the model from which we can perform either one *overflow* up until at most four *overflow*s simultaneously. We require the four alternatives, since the four buffers can potentially all overflow simultaneously. Furthermore we abstract from all other actions besides *overflow*, since the specified multi-actions in the modal formula must exactly match the transitions performed by the model. The abstraction is defined by the following model:

init $\tau_{\{enter,leave,comm\}}(\partial_{\{send,recv,grant,free\}}($
$\Gamma_{\{send|recv|grant\rightarrow comm\}}(\Gamma_{\{send|recv|free\rightarrow comm\}}($
$Input(0,[]) \parallel Input(1,[]) \parallel Output(0,[]) \parallel Output(1,[])))));$

This model is used to verify modal formula (2).

3.5.3 Absence of Colliding Packets

The property that no simultaneous packet transfers are possible to the same output buffer is specified by the following modal μ-calculus formula:

$$\forall_{p,q:Packet}\forall_{i,j,k:\mathbb{N}}\ [true^{\star}.(comm(i,j,p) \mid comm(k,j,q))]false \tag{3}$$

The modal formula states that there exists no reachable state from which we can perform two packet transfers two the same output buffer.

This formula must be checked on the model after abstraction from all other actions besides the *comm* action. We perform the abstraction to prevent the specification of all the combinations of *enter* and *leave* actions that potential coincide. This means that for the Simplified Switch we use the following model:

init $\tau_{\{enter,leave\}}(\partial_{\{send,recv,grant,free\}}($
$\Gamma_{\{send|recv|grant\rightarrow comm\}}(\Gamma_{\{send|recv|free\rightarrow comm\}}($
$Input(0,[]) \parallel Input(1,[]) \parallel Output(0,[]) \parallel Output(1,[])))));$

In a similar way we define these abstractions for the Original and Modified Switch.

It is not allowed to send two interesting packets simultaneously. Hence we specify the formula: "For all reachable states it must not be possible to simultaneously transfer two packets, if both packets are interesting." So the following modal μ-calculus formula is constructed and subsequently verified:

$$\forall_{p,q:Packet}\forall_{i,j,k,l:\mathbb{N}}\ (int(p) \wedge int(q)) \Rightarrow [true^{\star}.(comm(i,j,p) \mid comm(k,l,q))]false \tag{4}$$

Again we need to abstract from all actions except the *comm* action, since the performed multi-actions must exactly match the prescribed multi-actions in the modal formula. Note that Requirement 3 is relevant to all of the specifications mentioned in this chapter. Requirement 4 is only relevant to the latter two models.

Requirement	Simplified	Original	Modified
1	✓, 3.550s	✓, 5m03.863s	✓, 5m16.921s
2	✓, 3.729s	✓, 7m35.686s	✓, 7m35.202s
3	✓, 3.778s	✓, 4m44.647s	✓, 4m49.101s
4	–	✓, 5m29.906s	✓, 5m39.844s
5	✓, 3.301s	✓, 4m22.232s	✓, 4m33.786s

Table 3.1 Verification results for five modal properties

3.5.4 Maximal Progress

A property we would like to verify is *maximal progress*. In the context of this case study, the property is phrased as: "It is impossible to transfer a single packet from an input buffer to an output buffer in case a simultaneous packet transfer is possible." A modal μ-calculus formula that captures this (provided that it is checked on the model after abstraction of all the actions except the *comm* action) is the following:

$$\forall_{p,q:Packet}\forall_{i,j:\mathbb{N}}\ ([true^{\star}](\langle comm(i,dest(p),p) \mid comm(j,dest(q),q)\rangle true \Rightarrow ([comm(i,dest(p),p)]false \wedge [comm(j,dest(q),q)]false))) \quad (5)$$

Note that modeling maximal progress, as done in Requirement 5, does *not* enforce that internal packet transfers take priority over external packet transfers.

3.5.5 Verification Results

The requirements have been checked for all the (relevant) specifications. All of the buffers in the specifications have the buffer capacity of size 3. This capacity has been chosen because it still allows for a reasonably fast analysis. The analysis has been conducted with the mCRL2 toolset (Release 2010, January), on an x86-64 GNU/Linux, running kernel 2.6.31.12, with an Intel© Core™ 2 Duo Mobile Processor T9600, and 4GB of RAM.

The results of the formal analysis are captured by Table 3.1. Requirements that hold, w.r.t. a particular specification are marked with a "✓". The (user) time it took to perform the different verification runs are indicated as well. Requirements that are irrelevant for a specific model are marked with a "-". It shows that for all of the models all relevant formulas hold. All analyses have been performed without any attempts to reduce the verification timings with the help of state space reduction techniques.

3.6 Comparison to Other Specification Languages

This case study originates from the work described in [DS09]. There, the authors discuss the same case study for the specification languages: TLA+, Bluespec, Statecharts and ACP. This section describes the comparison of the mCRL2 language to the other

languages. We focus on the same three aspects as the authors of [DS09], namely the locality of reasoning (Chapter 3.6.1), the adaptability of the language (Chapter 3.6.2) and maximal throughput (Chapter 3.6.3). Furthermore we extend the scope by taking verification into account (Chapter 3.6.4).

Before explaining the comparison, we provide a brief description for each of the four languages. Firstly, TLA+ (the Temporal Logic of Actions) [Lam02] is a complete specification language that uses logic for the specification and reasoning about concurrent and reactive systems. It is designed for writing specifications consisting of non-temporal mathematics with temporal logic and tries to capture a complete system in a single formula. Secondly, Bluespec [HA00] is a guarded command language, based on an operation-centric description, where the behavior of a system is described as a collection of atomic operations in the form of rules. These rules are defined by a predicate condition and the effect on the state of the system. During execution several rules are concurrently executed in a clock cycle, thereby performing its execution. Thirdly, we consider Statecharts, which are an extension of conventional state-transition diagrams extended with three concepts. These concepts are hierarchy, concurrency and communication [Har87]. The graphical and hierarchical presentation enables engineers to adapt to the required level of detail for a system. Finally, the comparison covers the Algebra of Communicating Processes (ACP) [BK84]. ACP is an algebra for specifying and manipulating the behavior of models. It facilitates the behavioral description for non-deterministic choices, sequential operations, parallel composition, deadlock and communication.

3.6.1 Locality of Reasoning

Every system that is built from components, has a localized per component view that specifies its individual behavior. By means of some form of communication they exchange information. Hence, the authors of [DS09] perceive that local reasoning, the way that expresses local behavior, is a realistic (and subjective) attribute of a specification language. Within this case study we 'measure' the locality of reasoning from the way in which priorities are assigned to the routing of packets.

To reduce the amount of global reasoning w.r.t. the communication, we have generalized from the specific implementations of the input buffers. This allows us to reason on a local level about priorities. If we compare our models to those given in ACP, we see that by modeling priority as permissions, we abstract from the contents of the buffers such that they become invisible to other processes. In the given ACP models, the buffers are directly inspected by the other processes. This requires a more spatial reasoning in ACP to derive priority.

Within the TLA+ language, the priority of a packet transfer is dealt with on a local level. So w.r.t. assigning priority to executing actions, the mCRL2 language and the TLA+ language are comparable. We do note that the input buffers, as well as the output buffers are grouped in the TLA+ language. This enables for TLA+ actions to directly observe the buffer of another process at a local level. When comparing this method to the one specified in our models, we believe that it is possible in the

mCRL2 language to express the same behavior. However it would require an additional (global) process that mimics the behavior and controls the transfer. Hence, this would lead to a design that is more spatial, since we need to model the intended semantics explicitly, which are covered by the implemented semantics of the TLA+ language.

The Bluespec specification defines rules that implicitly deal with mutually exclusive access to shared resources. When multiple rules access a same resource, access is given to the rule defined first in the priority hierarchy. By applying this technique, they ensure priority among packet transfers. Note that priority rules are defined on a spatial level. Therefore, the reasoning is more spatial than the one used in mCRL2 language.

Within Statecharts all behavior of the buffers is locally specified. However global temporal reasoning is required to establish the priority among packet transfers. A simultaneous transfer requires a global spatial reasoning over at least four individual Statecharts i.e., the different buffers.

3.6.2 Adaptability

The authors in [DS09] only explain the TLA+ language for the simple switch. Though they claim that TLA+ relates to Bluespec, they do not show the models for the original and modified switch. For that reason, the adaptability of the TLA+ language is unclear, since we are no experts in it. This does not permit us to judge whether the mCRL2 language performs better or worse in terms of adaptability.

A similar reasoning holds for Statecharts. The authors describe in a fairly easy way how to model a simple switch from the original switch. However, in the subsequent discussion they show that the presented model of the original switch is incorrect and requires a more complicated model to capture the design intent. Since the corrected, and more complicated models are not given, it is not fair to make comparisons.

For modeling the modified switch in the Bluespec specification language, the authors require an entire redesign of the original switch, such that all priorities are defined by separated rules. This leads to almost a duplication of the model. As we compare the same extension for our modified switch in the mCRL2 language, we only have to split a summand and alter a guard, which are rather small modifications.

ACP serves well in terms of adaptability for this case study. As mCRL2 falls in the same category as ACP, this also holds for mCRL2. Therefore in terms of adaptability, mCRL2 and ACP are similar.

Furthermore, we have set up the processes of the buffers in such a way that they could be reused in a more general specification, e.g., an $N \times M$ specification. To do so, we are required to add extra parallel process references in the initialization, and add extra rules to the data equations for routing packets. Within the current models we allow that only one packet is sent simultaneously per clock cycle. By adding more processes, this bound does not change. To increase the throughput, e.g., allowing more message transfers per clock cycle, we need to add summands that grant this communication. We argue that these modifications could be done at a local level. As

the intended semantics for the internal communication of an $N \times M$ specification are not clearly defined, we cannot provide the specifications in detail.

3.6.3 Maximal Throughput

Within the specification maximal throughput is achieved by executing multiple actions in a single clock cycle. Therefore, this comparison narrows down the scope to the behavior of simultaneous actions.

It is not possible to describe the simultaneous transfer of packets in the TLA+ language and ACP. Therefore an engineer is required to apply a spatial reasoning to verify that packets are indeed simultaneously transferred. As we compare the formalism to the mCRL2 formalism, we see that within the mCRL2 language, it is possible to define multi-actions. We believe that multi-actions are a more suitable notion to specify the throughput behavior as it better relates to the concept of simultaneous packet transfer. Therefore it is not necessary for an engineer to reason about multiple transitions that represent a simultaneous transfer.

For Bluespec specifications, a greedy run-time scheduler tries to acquire maximal throughput. It should be noted that in some cases a maximal throughput cannot be obtained, even though all conflict-free rules are selected. To minimize latency, the scheduler may choose a maximal set of actions that are executed in every (hardware) clock cycle. Therefore it is possible that this set violates the maximal throughput requirement [SS08]. As exploration in the mCRL2 toolset is performed exhaustively, and latency is no issue, maximal throughput is guaranteed, by means of synchronizing actions and guards. Furthermore, although not specified here, we believe that it possible to use the mCRL2 time operator to enforce throughput in different ways, e.g., by enforcing the execution of actions at predefined timestamps.

Regarding Statecharts, the authors of [DS09] do not provide a suitable description in their paper, as they specified a wrong model. Therefore a comparison for maximal throughput, renders useless as the throughput analysis on the Statechart specifications are omitted. Note that this does not mean that it is impossible to give a correct model using Statecharts.

3.6.4 Verification

The authors of [DS09] are unable to convince themselves that the specifications they give are correct w.r.t. the design intent. As the remark essentially holds for all specifications, it already shows the first pitfalls in concurrent system design.

In line with the authors of [DS09], we agree that global reasoning is required (i.e., over the entire model) to verify system requirements. This however is a difficult task. As the description of the models is fairly simple, their explicit behavior is not. In Figure 3.2* we have taken the opportunity to show, that even for a small system like

*The numbers for the model of the modified switch are omitted as they are the same for the original model.

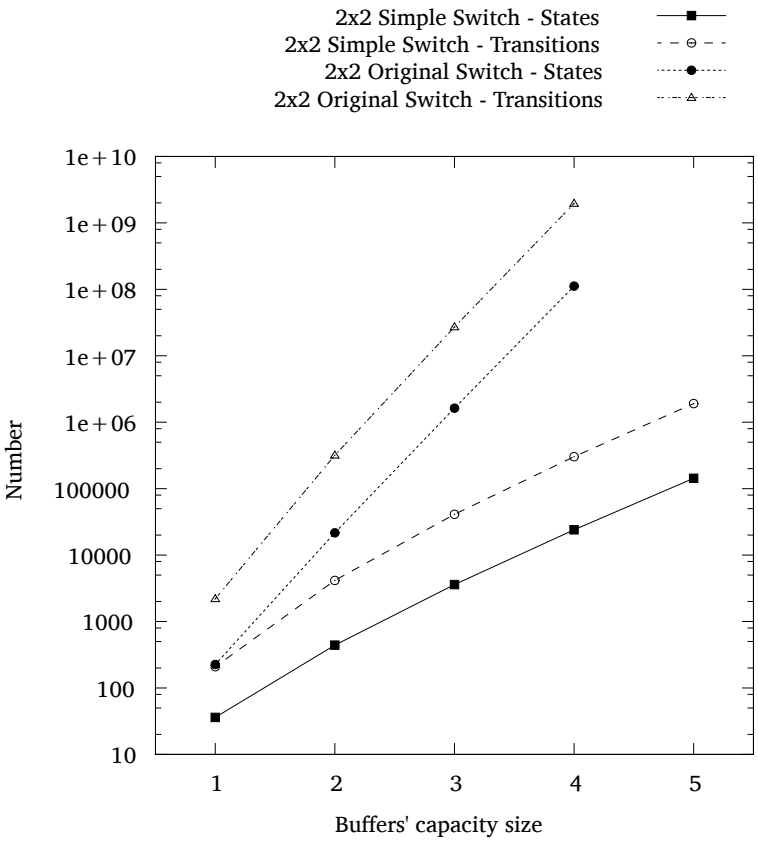


Figure 3.2 Complexity of the mCRL2 specifications expressed in the number of states and transitions for the simplified and original switch models

the simplified switch, we already specified a system for which the behavior cannot be overlooked by human reasoning[†]. For a buffer capacity of three elements, we generate a state-space of 3600 states with 41137 transitions.[‡] Nevertheless, with the automated methods of the mCRL2 toolset is possible to verify interesting properties on modeled systems.

[†]These numbers are obtained, without applying any reduction techniques. We are aware that these numbers could be reduced. The number of states and transitions are given on a logarithmic scale.

[‡]Note that multi-actions that contain precisely the same actions are only taken into account once. Otherwise, the numbers of transitions would have been much larger.

3.7 Conclusions

This chapter shows with the help of a case-study that the mCRL2 language and its toolset are suitable to model and subsequently analyze system descriptions in which multi-party communications are combined with priority-based communications. We have tried to apply local reasoning as much as possible, by generalizing the behavior of the buffers by type. Thereby we preserve both the possibility to transfer prioritized as well as simultaneous packets. Furthermore, we have shown that the mCRL2 formalism is capable to verify some behavioral properties. This increases confidence that the models represent the design intents. From the modeled system descriptions, we may conclude that the mCRL2 formalism is at least comparable to the formalisms used in [DS09], and is in some cases even more suitable to specify complex system designs.

Note that the comparisons are based on subjective grounds. For a fair comparison, one should study the possible language constructs for each of the formalisms and point out the differences. This requires an expert over the involved formalisms or a cooperation among experts. Since the case study is centered around a specific specification, for which models are created according to the level of the expertise of the engineers, the outcome of the comparison is subjective. As we consider ourselves experts, when it comes to the mCRL2 specifications, and are familiar with ACP and Statecharts, we are confident about the claims made between these formalisms.

We have shown that it is possible to capture relative performance requirements, without explicitly stating time. Since the mCRL2 languages falls into the category of timed process algebras [BM02], it allows engineers to also specify real-time behavior. Nevertheless, we have chosen not to do so for several reasons. Firstly, we would like to have a fair comparison between the untimed formalisms. Secondly, timed requirements tend to be more complex in general, and require challenging manipulations on the mCRL2 specifications before one can verify requirements. Nevertheless, we believe that the case study that is considered in this chapter can be formulated in a timed setting, or serve as a subject of study for reduction and analysis techniques for timed systems.

Chapter 4

Modeling Implementations

4.1 Introduction

The behavior of complex systems is controlled and steered by controllers. These controllers distribute tasks for execution, facilitate communication, and act as a watchdog over the behavior executed by different components. While systems tend to get more complex, in both the number of components as well as the corresponding behavioral execution, the safety governed by the controller becomes more vital. To govern the system's behavior, the number of lines of code for the software controller grows as well. To ensure that shipped systems are fault-free tests are performed. Unfortunately, the absence of errors is not guaranteed by only executing tests.

To increase the level of confidence on the executed behavior, formal methods can be used. This chapter derives a formal model from code. By observing the code we model the behavior that is *implemented* rather than the behavior that is *intended* or *desired*. Hence, we apply a method that is commonly proposed by industry, namely constructing a model from an existing implementation. The route describes an instance of a bridge between the execution environment and the analysis environment from Figure 1.1.

Deriving a model from source code is a challenging task. Source code contains many implementation details that are relevant for execution, however irrelevant for what should be analyzed. Besides, if the code for any realistic application would be directly translated into a behavioral model, it should result in a specification that is far too complex to verify any requirement. Therefore, as the requirements are (often) stated at a more abstract level, we need to perform an abstraction such that the behavior and the requirements are described at the same level.

This chapter shows a method that has been applied to analyze functional requirements in an industrial setting. The requirements are stated in terms of the communication between processes. Moreover, all of the interesting requirements are safety

requirements [Lam77]. Therefore we apply two abstractions, namely (i) we abstract from all internal actions, and (ii) we abstract from all data (i.e., all values of variables and all assignments). By performing the abstraction mentioned in (i), we create a setting where conditions cannot be evaluated accurately. As a result we perform (ii), where we replace all conditions by non-deterministic choices. This creates an over-approximation of the system's behavior, since the model is less restrictive, with respect to the actual implementation. In turn, this preserves a simulation relation [GG89]. Note that the abstraction affects the verification results. This means that when a safety property holds in the over-approximation, it must hold for the real system. On the contrary, when a safety property does not hold for the over-approximation, it may still hold for the real system. To determine whether a safety property is indeed violated by the system, it requires the actual judgment of an engineer.

A direct transformation of the actual code into mCRL2 specifications is possible, however it renders any exhaustive verification techniques useless. Because the implementation details introduce such a tremendous amount of information, it leads to models that dictate behavior that exceed the resource limits of the current state-of-the-art analysis techniques. Therefore we need to abstract from them. With the help of a case study, we assess the feasibility of such an approach. Hence we take a controller that is actually used in a prototype printer. The functionality is specified by tasks which facilitate a structured way of executing the code that drives the components of the system. We assume that the source code for the controller is written in a Simplified Concurrency Programming Language (SCPL, Chapter 4.3). SCPL in itself is an abstract representation from the actual source code, that expresses the concurrent behavior found in the imperative program, while preserving its characteristic language constructs. To verify safety requirements, we transform a model written in SCPL to an mCRL2 model that in turn is verified using the mCRL2 toolset. To demonstrate the practical value of the method, the transformation from a SCPL model to an mCRL2 model has been performed by hand.

This chapter is structured as follows. Chapter 4.2 provides a background on the modeled system. Chapter 4.3 introduces the SCPL language. Chapter 4.4 relates the architecture of the system to the SCPL language. In Chapter 4.5 we describe the abstraction technique that transforms a model written in the SCPL language to an mCRL2 model. The verified safety properties and their results are discussed in Chapter 4.6. Related work and conclusions are respectively discussed in Chapter 4.7 and Chapter 4.8.

4.2 System Description

The source code that has been used for this case study originates from an industrial system, called "Lunaris" [Roo07]. Lunaris is an Etch Resist Printer for manufacturing Printed Circuit Boards (PCBs). In current PCB production processes, the substrate is laminated with a photo resist and uses a lithographic process to create the desired photo mask on substrates. With the development of Lunaris, it is possible to skip the

expensive task of creating a mask. By directly printing the resist in a desired pattern, it is possible to create customized and individual PCBs at lower costs.

The prototype printer has been developed and extensively tested for over one year. The system has many physical components, but we limit ourselves to the (behavioral) requirements at the level of the (software) controller. At the controller level, Lunaris consists of 245 multi-threaded implemented tasks (running in parallel). The implementation is written in the C# language [AW02]. The tasks specify behavior that includes printing, moving physical components, logging and error handling. In total 170.000 lines of code are implemented to specify the controller's behavior. The code is distributed over 120 classes in 40 files. Note that we do not model the control flow of the controller. As a result, the controller is free to execute any behavior it wants. The actual behavior of the controller is programmed by an outsourced party.

In this chapter the requirements are derived from the system code. The code contains a special section that states the requirements in terms of the actual implementation. These requirements are run-timed checked, for which they have specified two kinds of safety requirements. The first set of requirements raises a warning whenever they are violated. The second set of requirements brings the system to an immediate halt whenever they are violated. These rules are specified in a separate monitor process, thereby acting as a watchdog over the executed code. The verification results are used to inform the external party, the ones responsible for programming the control flow of the controller, on the behavior that may not be executed*.

4.3 Simplified Concurrency Programming Language

To illustrate the approach we introduce the *Simplified Concurrency Programming Language* (SCPL). The SCPL is a programming language deduced from modern imperative software programming languages. The language enables users to specify parallel programs, because it has notions to express concurrency, but hides concepts that facilitate object oriented software development (e.g., definition of classes, creation of objects, ...) or memory management (allocation of variables, pointer dereferences, ...).

The grammar of SCPL is described by the following BNF:

$$
\begin{array}{lcl}
\langle \texttt{program} \rangle & ::= & \langle \texttt{process} \rangle \ \langle \texttt{program} \rangle \mid \langle \texttt{process} \rangle \\
\langle \texttt{process} \rangle & ::= & \textbf{proc}\ C = \langle \texttt{statement} \rangle\ \textbf{return} \\
\langle \texttt{statement} \rangle & ::= & \textbf{call}\ N \mid \textbf{x} := \textbf{e} \mid \langle \texttt{statement} \rangle ; \langle \texttt{statement} \rangle \mid \\
 & & \textbf{if}\ b\ \textbf{then}\ \langle \texttt{statement} \rangle\ \textbf{else}\ \langle \texttt{statement} \rangle\ \textbf{fi} \mid \\
 & & \textbf{while}\ b\ \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \mid \textbf{do}\ \langle \texttt{statement} \rangle\ \textbf{od} \mid \\
 & & \textbf{suspend} \mid \textbf{resume}\ N
\end{array}
$$

Every process consists of a unique identifier, i.e., the process identifier, and a body: a process with process identifier C and body of statements S is denoted by **proc** $C =$

*This chapter only describes the publicly available parts for the models and code. Detailed information on both the code and corresponding models are made available under a non-disclosure agreement.

S **return**. The analysis assumes that a program consists of at least one process and that the process that is initially active corresponds to the process labeled with the identifier *init*. The body of a process consists of statements that denote calls to other processes **call** N (where N is a non-empty set of process identifiers), multi-assignments $\mathbf{x} := \mathbf{e}$, sequential compositions $S; S'$, the conditional execution of statements **if** b **then** S **else** S' **fi**, guarded repetitions **while** b **do** S **od**, unguarded repetitions **do** S **od**; and the statements **suspend** and **resume** N, respectively denoting the pause of a process and the continuation of a non-empty set of processes. As in many imperative programming languages, we assume that the constructs are defined by their intended informal semantics.

The corresponding formal semantics of the syntax is omitted. The second part of the thesis describes a method that derives a formal model with the help of the operational semantics. Therefore, one could argue that we could have taken this case, provide its formal semantics and compare the alternative route. As we take a more comprehensively case study, namely the semantics of the mCRL2 language, we feel that showing the route for SCPL is redundant.

4.4 Relating the Implementation to SCPL

The code of the controller hides low-level implementation details by a proprietary framework. The framework groups these implementation statements and represents them by *tasks*.

Tasks express different activities that need to be performed by the system. They are used to operate hardware, log information, perform error handling, delay execution, ignore errors, etc. To control the system a task may execute its implementation, select a task for execution if a guard holds, or start the concurrent execution of several other tasks. To accomplish the communication between the tasks, the tasks execute their behavior via a master-slave protocol. That is:

- A task is a master if the task executes other tasks.
- A task is a slave if the task is executed by another task.

Note, that tasks that are slaves can be masters for other tasks. While different tasks are executed concurrently, their underlying statements are interleaved during execution. The system executes only one statement at a time. Moreover, it is not possible to run a task that is already running. It is considered to be illegal behavior, if a task is running, and another task wants to start the running task.

To signal other tasks for execution, tasks communicate over non-lossy channels. The communication consists of four different messages:

Start A master wants to start a task of a slave.

Done A slave indicates that a task has been successfully terminated and returns the control back to its master.

Resume A master wants to resume a task of a slave.

Suspend A slave suspends the current process and notifies its master.

While we are only interested in safety properties, we assume the following:

1. Tasks that are exclusively used for logging and error-handling local to components are considered to be irrelevant.
2. The system prescribes "good weather" behavior. "Good weather" behavior assumes that the components behave without any faults. This means that a printhead is not broken, the system prints when it is supposed to, communication channels are non-lossy, etc.
3. The protocols that facilitate communication are handled correctly by the framework and the embedded software is implemented according to the specification. By this assumption we do not have to specify and verify the software that provides the communication and the software on the embedded systems.
4. Time is not modeled explicitly. The behavioral correctness of the controller should not be affected by the required amount of time to perform a task. This decisions prohibits the verification of performance properties.
5. In the initial state of the implementation, the system is turned off and all components are positioned such that they reside in their initial position.

After applying these simplifications, we obtain 236 tasks, that are executed concurrently. While studying the behavior of these tasks, we see that their behavior can be categorized into two types, namely *execute tasks* and *switch tasks*.

4.4.1 Execute Tasks

An *execute task* is a task that is executed once, e.g., moving the print head device to a given position. When started, an execute task automatically completes after a finite amount of processing time.

The semantics of an execute task is depicted by a hierarchical state machine, as illustrated in Figure 4.1. A sub-task is indicated by a rectangle. A single lined box indicates that a sub task consists of a single state. A double lined boxes indicates that a sub task is a compound state. The behavior that is executed in the compound state, is dictated by the state machines that are included within.

An execute task has two stages, namely *Idle* and *Executing*. A task is idle when no statements need to be executed. A task is executing when it needs to execute statements. If a task finishes the execution of all statements it returns to the idle state, i.e., it successfully terminates.

The behavior of an execute task with identifier C is mapped to an SCPL process of the form:

$$\textbf{proc } C = \text{“Executing”} \ \textbf{return}$$

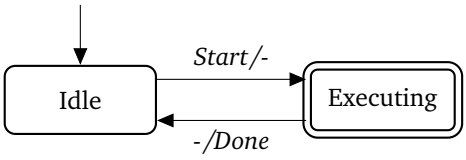


Figure 4.1 State diagram for an Execute Task

4.4.2 Switch Tasks

A *switch task* is a task that whenever started, needs to be stopped explicitly. Switch tasks are often used to enable hardware components, e.g., to enable controllers when the system reaches a certain run-level.

The semantics of a switch task is depicted as a hierarchical state machine, which is illustrated by Figure 4.2. A switch task has four stages, namely *Idle*, *Executing*$_1$, *Executing*$_2$ or *Enabled*. The first two stages are similar to the ones of the execute task. A process is *Enabled* (a.k.a. temporarily idle), when the process suspends itself after executing some statements, although some statements are left for execution at a later moment in time. We assume that only *Enabled* tasks can be suspended. This means that when a task is *Idle*, it cannot be resumed or started by a resume.

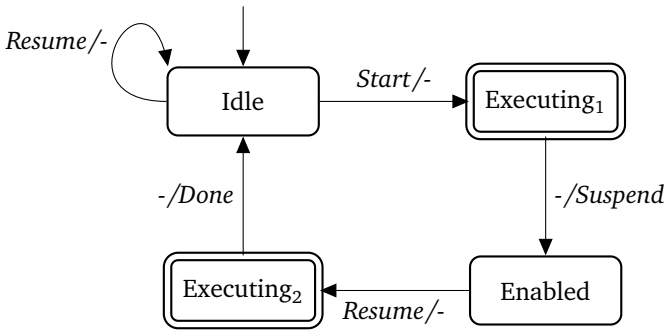


Figure 4.2 State diagram for a Switch Task

The behavior of a switch task carrying the identifier C is mapped to an SCPL process of the form:

$$\textbf{proc } C = \text{“Executing}_1\text{”; } \textbf{suspend}\text{; Executing}_2\text{” } \textbf{return}$$

4.5 Transformation Scheme

The transformation scheme describes a transformation function that takes a program written in the SCPL language and produces an mCRL2 specification. The transformation function is described by function $\mathcal{A}$. The resulting mCRL2 specification consists of an initial process and a set of mCRL2 process equations.

The transformation function does not cover the required sort declarations, the required action declaration, nor the communication among processes. Although they are required by the mCRL2 specification, we specify them separately a priori.

So, we define **sort** P_D that models all process identifiers. Then for each process with identifier C in the SCPL program, including the *init* variable, we specify a constructor **cons** $C : P_D$. Moreover, we introduce for every C a recursion variable X_C in the mCRL2 specification.

For the transformation we require (and define) the following actions:

- $Start_s : P_D$ denotes the request to start process $C \in P_D$;
- $Start_r : P_D$ denotes the notification of the request for starting the process $C \in P_D$ (invoked by another process);
- $Done_s : P_D$ denotes the successful termination of process $C \in P_D$;
- $Done_r : P_D$ denotes notification of termination for a run of process $C \in P_D$;
- $Suspend_s : P_D$ denotes the suspend of process $C \in P_D$, activated by task C itself. If a process decides to suspend its execution, it sends the calling process a suspend signal. This allows the calling process to continue;
- $Resume_r : P_D$ denotes the notification of the request to resume the process $C \in P_D$;
- $Resume_s : P_D$ denotes the request to resume the suspended process $C \in P_D$.

We introduce the actions $Start, Done, Suspend, Resume : P_D$ that denote the resulting communicating actions for the corresponding send/receive requests. Assuming that the name of the initial process is indicated by *init*, we define the transformation function $\mathcal{A}$ as:

$$\mathcal{A}(p_1, \ldots, p_k) = (\partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \parallel (\|_{C \in P_D} X_C)))), \bigcup_{i=1}^{k} \mathcal{A}'_{\chi_i}(p_i))$$

where

- k denotes the number of processes and p_i denotes process i $(1 \leq i \leq k)$;
- $Bl = \{Start_s, Start_r, Done_s, Done_r, Resume_s, Resume_r, Suspend_s\}$ expresses the set of blocking actions, i.e., the actions that may not occur in isolation;
- $B = \{Start_s \mid Start_r \rightarrow Start,\ Done_s \mid Done_r \rightarrow Done\}$ denotes the set of primitive communications, i.e., the actions that need to communicate;

- $E = \{Suspend_s \mid Done_r \rightarrow Suspend,\ Resume_s \mid Resume_r \rightarrow Resume\}$ specifies the set of additional communications, i.e., the second set of actions that needs to communicate;

- $\|_{j \in J} X_j$ describes the processes that are running in parallel, that is recursively defined as:

$$\|_{j \in \emptyset} X_j = \tau, \qquad \|_{j \in J \cup \{k\}} X_j = X_k \parallel \left(\|_{j \in J \setminus \{k\}} X_j \right);$$

- the sets χ_i denote the sets of recursion variables that are used in the transformation result for the construct "**while** b **do** ⟨`statement`⟩ **od**" and the construct "**do** ⟨`statement`⟩ **od**". These process identifiers are pairwise disjoint and are disjoint from the set of recursion variables for the process definitions.

- $\mathcal{A}'$ denotes the transformation function for processes defined in Chapter 4.5.1.

The encapsulation operator ∂_{Bl} and communication operators Γ_E, Γ_B are applied to the parallel composition to model the communication between processes and to block individual non-successful communications. We specify the communication operators Γ_E and Γ_B such that we guarantee a unique solution, i.e., it is impossible to obtain two results given any input of actions. It is not used to specify any priority.

Every process that is defined in the SCPL specification is associated with at least one mCRL2 process equation by means of the transformation function $\mathcal{A}'_{\chi_i}$, namely X_C corresponds to the translated process p labeled with identifier C. The other variables χ_i (also included in P_D) are introduced to capture repetitions in the body of a process. To ensure that the introduced recursion variables differ from other recursion variables, the transformation function is parameterised with χ_i, which are free to be used recursion variables and are chosen sufficiently large.

We assume that the initialization process *init* is called once from outside the system.

4.5.1 Processes

Processes decompose the system's functionality into smaller manageable parts. Every process carries out a specific task. The behavior of a process is often implemented by a function, subroutine, procedure or some functional behavior. The behavior of an individual process is defined by statements placed in some order. Let **proc** $C = S$ **return** denote the implementation of a process, where C defines the process identifier and S defines the statements placed in some order.

To model the SCPL process as an mCRL2 specification, we apply function $\mathcal{A}'_{\chi}$ to the implementation. Here χ denotes the set of available recursion variables. The mCRL2 process equation that is provided has two summands. The first summand reflects the start, the normal execution and the termination of a task. The second summand models the behavior that is associated to resuming an idle task. Hence, no statements are executed. The transformation function uses the identifier C to indicate

the translated process. The identifier is used by other processes to execute the tasks, i.e., to become enabled.

The transformation function for process identifier C and statement S is given by:

$$\mathcal{A}'_{\chi}(\textbf{proc } C \ = S \textbf{ return}) = \left\{ \begin{array}{rl} X_C = & \mathit{Start}_r(C) \cdot t_s \cdot \mathit{Done}_s(C) \cdot X_C \\ + & \mathit{Resume}_r(C) \cdot \mathit{Done}_s(C) \cdot X_C \end{array} \right\} \cup E_s$$

where $(t_s, E_s) = \mathcal{A}''_{\chi,C}(S)$ and $\mathcal{A}''_{\chi,C}$ are the transformation functions for statements (Chapter 4.5.2).

4.5.2 Statements

This subsection discusses the transformation of statements performed by the function $\mathcal{A}''_{\chi,C}$. All occurrences of p and q denote (process) statements. All occurrences of b are Boolean ($\mathbb{B}$) expressions.

Interface Calls An interface call contains a non-empty set of process identifiers. If a set contains one element, it behaves as a call to a single process. If a set contains more elements, it behaves as a call to multiple processes. A call simultaneously enables the start of the processes referred to by the set of identifiers N. These processes are then executed concurrently. Processes can only be started by an interface call when they are in the idle stage. If a call is addressed to a busy process or a temporarily idle process, the call is postponed until the process becomes idle. This implies that all processes need to be idle to conduct an interface call to multiple processes.

An interface call statement is translated by:

$$\mathcal{A}''_{\chi,C}(\textbf{call } N) = \left(\left(\Big|_{n \in N} \mathit{Start}_s(n) \right) \cdot \left(\Big|_{n \in N} \mathit{Done}_r(n) \right), \emptyset \right)$$

where $\Big|_{n \in N} \alpha(n)$ is inductively defined as:

$$\Big|_{n \in \emptyset} \alpha(n) = \tau, \qquad \Big|_{n \in N \cup \{k\}} \alpha(n) = \alpha(k) \mid \Big|_{n \in N \setminus \{k\}} \alpha(n).$$

Since no additional process equations are introduced, the right element of the tuple is empty.

We assume that all calls are made to idle processes. Therefore postponing a call conflicts with the assumption. This type of behavior is considered illegal (Chapter 4.4). To assert that no illegal behavior is eliminated, we verify that it is not possible that a finish and a start of a task are subsequently executed. This requirement is verified in Chapter 4.6.3.

Assignments The multi-assignment statement $\mathbf{x} := \mathbf{e}$ defines the (atomic) value updates for the variables $x_1, \ldots, x_n$ with the values of $e_1, \ldots, e_n$. Because we already decided to abstract from all data, the values, variables and corresponding assignments become irrelevant. So we transform a multi-assignment by:

$$\mathcal{A}''_{\chi,C}(\mathbf{x} := \mathbf{e}) = (\tau, \emptyset)$$

where the assignment is transformed into an internal non-observable action τ. Moreover, no additional process equations are introduced.

Sequential Composition The sequential execution of statements is described via the sequential composition operator. It is evident that the sequential order in which the statements of a task are executed stays preserved. Since the mCRL2 language knows a similar construction, the transformation for the operator is (almost) straightforward

$$\mathcal{A}''_{\chi,C}(s_1 \; ; \; s_2) = (t_{s_1} \cdot t_{s_2}, E_{s_1} \cup E_{s_2})$$

where

- $(t_{s_1}, E_{s_1}) = \mathcal{A}''_{\phi,C}(s_1)$, and
- $(t_{s_2}, E_{s_2}) = \mathcal{A}''_{\psi,C}(s_2)$

Here ϕ and ψ are sets of recursion variables such that $\phi \cap \psi = \emptyset$ and $\phi \cup \psi \subseteq \chi$. These sets need to be chosen large enough to allow for the subsequent transformations to provide enough fresh recursion variables for transforming both the statements s_1 and s_2. This means that for every recursion in the subsequent transformation there must be a least one unique recursion variable available.

Conditional Composition The outcome of an evaluation of a conditional composition operator depends on the values of variables. As we have chosen to abstract from the values of variables, it is impossible to determine the outcome of a condition. Therefore, we assume that the condition evaluates to either *true* or *false*. This implies that the conditional composition operator behaves as a non-deterministic choice.

As a result, we translate all conditional composition operators to non-deterministic choices by:

$$\mathcal{A}''_{\chi,C}(\textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}) = (t_{s_1} + t_{s_2}, E_{s_1} \cup E_{s_2})$$

where

- $(t_{s_1}, E_{s_1}) = \mathcal{A}''_{\phi,C}(s_1)$, and
- $(t_{s_2}, E_{s_2}) = \mathcal{A}''_{\psi,C}(s_2)$

Here ϕ and ψ are sets of recursion variables such that $\phi \cap \psi = \emptyset$ and $\phi \cup \psi \subseteq \chi$. Analogue to ϕ and ψ of the sequential composition, these sets need to be chosen large enough to allow for the subsequent transformations to have enough fresh recursion variables available.

Recursion Recursions are used to execute a set of statements that need to be carried out several times in succession. Recursions are either used for computational purposes (for which they are finite) or for dictating the control flow (being possibly infinite).

Recursions are modeled by means of recursion variables. When a control recursion is finite, it has a condition which determines when to abort the recursion. Under the assumption of fairness and the decision to abstract from all data, we *assume* that these kind of recursions terminate. We do not know *when* they terminate. So we model the conditional choice as a non-deterministic choice (as is the case for conditionals). For unguarded recursion there is no reason to introduce a non-deterministic choice.

The reason for explicitly having unguarded recursion in SCPL is that virtually all systems have parts that need to run continuously during execution. In these circumstances it must not be possible to abort the recursion.

Consequently, we provide two transformation functions. The first transformation is defined for the unguarded recursion. The second transformation is defined for the guarded recursion.

$$\begin{aligned}
\mathcal{A}''_{\chi,C}(\textbf{do } p \textbf{ od}) &= (Y, \{\, Y = t_p \cdot Y\} \cup E_p) \\
\mathcal{A}''_{\chi,C}(\textbf{while } b \textbf{ do } p \textbf{ od}) &= (Y, \{\, Y = t_p \cdot Y + \tau\} \cup E_p)
\end{aligned}$$

Here Y denotes a recursion variable taken from χ, and t_p and E_p are defined as $(t_p, E_p) = \mathcal{A}''_{\chi\setminus\{Y\},C}(p)$. Note that an additional τ is required to end a guarded recursion.

Processes Suspension When processes are concurrently executed, it is often desired to temporarily suspend a process. An example could be that the process has to wait for another process to complete a certain task. For this reason the SCPL language facilitates a statement to that self-pauses an executing process. We assume that if a process performs a self-pause, it is eventually resumed. This choice is reflected in the transformation by modeling the self-pause and the resume by two sequentially executed actions.

$$\mathcal{A}''_{\chi,C}(\textbf{suspend}) = (\mathit{Suspend}_s(C) \cdot \mathit{Resume}_r(C), \emptyset).$$

Process Continuation Processes that have performed a self-pause stay temporarily idle, until another process signals the process to continue. The solution offered by the SCPL language, enables the continuation of a suspended process by means of a resume action.

Like interface calls, multiple processes are (concurrently) resumed by a single resume. We assume that a process that performs the signaling transfers the control to the resuming processes. The control stays with these processes until all of the resumed processes either terminate successfully and/or become temporarily idle.

The resume statement is translated as follows:

$$\mathcal{A}''_{\chi,C}(\textbf{resume } N) = \left(\left(\Big|_{n\in N} \mathit{Resume}_s(n)\right) \cdot \left(\Big|_{n\in N} \mathit{Done}_r(n)\right), \emptyset\right).$$

4.5.3 Transformation by Example

To illustrate the transformation we consider a system that has two concurrent processes *init* and P. Process P consist of two parts, separated by a suspend. *init* calls P and waits until P finishes the first part. Then process P finishes the first part, process *init* continues by resuming the execution of the second part of process P. The execution in the SCPL language is reflected by Algorithm 1:

Algorithm 1 A code snippet from the SCPL language

```
proc init =
    call P;
    resume P
return

proc P =
    b := true;
    suspend;
    b := false
return
```

After applying the transformation we obtain the following mCRL2 specification:

$$\begin{aligned}
\textbf{proc}\quad & X_{init} = Start_r(init) \cdot Start_s(P) \cdot Done_r(P) \cdot Resume_s(P) \\
& \qquad \cdot Done_r(P) \cdot Done_s(init) \cdot X_{init} \\
& \quad + Resume_r(init) \cdot Done_s(init) \cdot X_{init};
\end{aligned}$$

$$\begin{aligned}
\textbf{proc}\quad & X_P = Start_r(P) \cdot \tau \cdot Suspend_s(P) \cdot Resume_r(P) \cdot \tau \cdot Done_s(P) \cdot X_P \\
& \quad + Resume_r(P) \cdot Done_s(P) \cdot X_P;
\end{aligned}$$

$$\textbf{init}\quad \partial_{Bl}(\Gamma_E(\Gamma_B(Start_s(init) \cdot Done_r(init) \parallel X_{init} \parallel X_P)));$$

The corresponding LTS is depicted in Figure 4.3.

4.6 Verification

This section discusses the verified safety requirements. The requirements have all been stated as assertions in the code of the system's architecture. Because verification is performed on the mCRL2 model, we need to state the requirements in terms of the mCRL2 model, rather than the SCPL language. Thus the actions that are used in the modal formulas, are obtained from the translated mCRL2 specification.

Safety rules have been formulated in the system's architecture to ensure correct behavior. A safety rule is a condition that may not be violated by the execution of the system. Lunaris specifies two kinds of safety requirements. The requirements

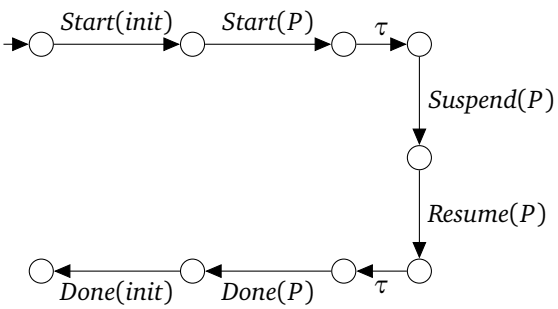


Figure 4.3 LTS for an SCPL specification

mentioned in Chapter 4.6.1 and Chapter 4.6.2 have all been derived from the specified requirements in the implementation. The soundness of the model is discussed in Chapter 4.6.3. Chapter 4.6.4 describes the verification details.

4.6.1 Warnings

The first set of requirements consists of eight rules. If an implemented requirement is violated during execution, the controller only emits a warning, but continues the system's execution.

The first class of safety requirements are templates of the form: The "Switch Task (ST)" must be running if the "Execute Task (ET)" is executed. These properties are decomposed into the following modal formulas:

- An execute task ET may not be started before a switch task ST is "Enabled:"

$$\begin{array}{ll} & [(\neg Suspend(ST))^{\star} \cdot Start(ET)]false \\ \wedge & [true^{\star} \cdot Start(ST) \cdot (\neg Suspend(ST))^{\star} \cdot Start(ET)]false \end{array}$$

- An execute task ET may not be stopped after the switch task ST is being stopped:

$$\begin{array}{ll} & [true^{\star} \cdot Start(ET) \cdot (\neg Done(ET))^{\star} \cdot Resume(ST)]false \\ \wedge & [true^{\star} \cdot Start(ET) \cdot (\neg Done(ET))^{\star} \cdot Done(ST)]false \end{array}$$

The analysis shows that three of the eight safety rules are superfluous, i.e., a warning cannot arise in any of the behavior executed by the system.

4.6.2 Critical Errors

The second class of rules consists of 30 safety properties which only allow the execution of a task (T), if some precondition is met. These tasks, e.g., involve the

movement of the printhead calibration system or shuttle. Since they physically operate in each others workspaces, it is possible that the system incurs physical damage if these safety properties are violated. To prevent damage, the system halts when a precondition fails.

To verify the second class of requirements, temporal logic formulas of the following forms have been constructed (S, T, and U are actions):

$$[true^{\star} \cdot S \cdot (\neg T)^{\star} \cdot U]false$$

Informally the modal formula states, that there exists no reachable state from which it is possible to execute action S, followed by action U, such that all actions in between action S and U are all non T labeled actions.

All of the formulas have been checked and some requirements were violated in the model. Since the verification has been performed with a controller that is in no way restricted, the violating requirements had to be manually inspected, i.e., taking the restrictive behavior into account and simulate the code in a software in the loop environment, to rule out any false violations. It turned out that four requirements were potentially violating. As the actual implementation was performed by an external team of programmers, the illegal sequences have been communicated to prevent critical errors from happening.

4.6.3 Soundness of the Model

The translation of the system's code to an mCRL2 specification has been performed by hand. To ensure that all of the relevant code is translated, the translation has been performed in (small) incremental steps. By adding code, that is subsequently translated and added to the model, we could check the completeness by means of a deadlock analysis[†]. By performing the analysis we could see that any incomplete parts led to deadlocking states because of unsuccessful communications. By adding the missing behavior of the corresponding processes, and the introduction of successful communications, we were able to create a deadlock free model. This implies that (i) all used interfaces between the components are implemented in the code and (ii) we have modeled all the behavior that is executed by the controller.

Recall that for the abstraction we assume that no two same tasks could be executed simultaneously. Although it was assumed by engineers, there was no evidence that this assumption was valid. The behavior of a tasks is modeled by a single mCRL2 process and all processes are initially specified in a parallel composition. This implies that no two tasks, which carrying the same identifier, can be executed simultaneously. So, if a request to start a task would be sent to an already running task, the mCRL2 model would stall and postpone the request until the running task would have finished. Hence we would see the termination of a task, immediately followed by the start of a task. Therefore we verify the requirement that no two instances of a task carrying the same identifier are executed sequentially. If we find such a witness, it could

[†]The absence of deadlock is checked with the following modal formula: $[true^{\star}]\langle true\rangle true$

imply that a task had been stalled for execution. Therefore the following requirement has been specified and verified:

$$\left[true^{\star}\right]\left(\forall_{i:T}\left[Done(i)\cdot Start(i)\right]false\right)$$

where T denotes the set of identifiers for all switch and execute tasks. The verification revealed several witnesses that could all be traced to tasks that were sequentially executed in the code. While no other instances were found, it confirmed that no two same tasks could be executed simultaneously.

The engineers presumed that it would be possible to immediately resume a switch task, after it got suspended. This behavior would imply a faulty *Enabled* stage, since no behavior had been executed that could alter the internal state of the system. To eradicate the suspicion, we formulated and verified the following formula,

$$\left[true^{\star}\right]\left(\forall_{i:S}\left[Suspend(i)\cdot Resume(i)\right]false\right)$$

where S denotes the set of identifiers of all switch tasks. We performed the verification and did not find any witnesses. Therefore we could safely assume that no faulty *Enabled* stages were present.

4.6.4 Verification Details

The state space that coincides with the verification consists of 76256 unique states and 253145 transitions for the 236 tasks. The rather low amount of states results from the serialization and the dependencies between task. To verify the temporal formulas we needed to linearize the mCRL2 model into an LPS. For all the requirements, this linearisation step has been executed just once. This took approximately 53 minutes on a computer with an Intel® Pentium® D Processor 930 and 2 Gb RAM running Linux. The subsequent verification of a single requirement took less than 15 minutes. This includes the transformation of an LPS and a modal formula to a PBES, and the subsequent solving of the PBES.

4.7 Related work

To determine whether a system is free from programming bugs, inconsistencies, run-time errors, or contains non-portable constructs, various tools like LINT [Dar88, Joh78], POLYSPACE [Inc], and QA-C++ [PRQ] act as an extension to standard debuggers. When it comes to the verification of dynamic properties (deadlocks, undesired behavior) tools like Java PathFinder [Sofc] or StEAM [LME04] are used. These tools use a virtual machine in which models are translated to byte code. Afterwards they are executed to verify properties. Unfortunately, the size of the code is related to the underlying state space that needs to be explored. This means that if the size of the code becomes larger, it becomes harder, or even impossible to verify dynamic properties. As stated by Java PathFinder:

> "While software model checking in theory sounds like a safe and robust verification method, reality shows that it does not scale well."

To verify industrial systems, abstraction techniques are inevitable. One can argue that the work presented in this chapter is an implementation of the theory of abstract interpretation. In abstract interpretation [NNH99], abstract values are chosen for variables. Behavioral models obtained via this approach depend on the (initial) values of data variables. Consequently, it requires manipulation of the data variables. For relatively small systems, this method is fruitful. However larger systems may still face a state space explosion, due to the number of parallel processes combined with the number of possible abstract data values. To verify larger systems, either more coarse grained abstraction techniques are required (such as described in this chapter) or state space reduction techniques need to be applied (e.g., symmetry reduction, bi-simulation reduction, etc.). Since almost every task specifies unique behavior, we could not benefit greatly from symmetry reduction. This technique is found in [LN03, BDH01]. The application of bi-simulation reduction techniques requires the generation of the full underlying state space.

Related work can also be found in the Bandera tool set [CDH+00]. The Bandera tool set translates Java source code to a model, which verifies properties using model checking techniques. The Bandera toolset, only accepts closed code. For this reason the system needs to be fully implemented before it can be verified. With the help of extensions it is possible to verify open systems (e.g., an environment generator for Bandera [TDP03]), but it still requires a full and correct implementation of a source code unit. Since our method abstracts from variables we can deal with partly implemented units and code skeletons.

The author of [Kof07] presents a way for checking component behavior compatibility, written in behavior protocols and checked with the Spin model checker afterwards. Using LTL formulas, they manage to verify properties on a well documented system of 20 components. In our case study we tackled a bigger system running 230 concurrent processes, and performed a successful verification with a different tool set. Next to that the semantics of our components differs: we cope with processes that are suspended and need to be resumed afterwards, while the components mentioned in [Kof07] do not facilitate such a mechanism.

Work presented in [Hol01] describes a method that directly derives a Promela specification from C code. This technique creates for every command a corresponding action in a Promela specification. In [Web07] another approach is taken with Promela. Here experiments are conducted with a virtual machine based approach for state space generation. By evaluating the byte-code language, they provide a way to efficiently execute operational semantics for modeling and programming languages. Undoubtedly, these techniques perform well on small toy examples for examining specific code constructs. However when changing the scope from specific code constructs to the control flow for examining larger concurrent systems, more rigorous techniques are required. In that sense, the method described in this paper can be viewed as an extension to their techniques.

Notice that our work also shares resemblance with SLAM by Ball et al. [BR01]. One of the SLAM approaches is based on refining the abstractions (to rule out spurious counter-examples), and turns software implementations into Boolean programs [BR00]. The basic idea is to leave out data initially, and include it when needed later on. Data that is included in the refinement applies to variables that are used in conditions. With the help of a theorem prover and additional iterations for refinement the SLAM method tries to determine if it can solve the equations, thereby terminating the recursion. In rare cases, it is possible that the theorem prover used by SLAM cannot solve the equations, which leads to a non-terminating algorithm. Consequently, verifying safety requirements become impossible.

Counterexample-Guided Abstraction Refinement (CEGAR) (see [CGJ+03]) is an automatic iterative abstraction-refinement methodology for which a datapath abstraction results in an approximation of the original design, i.e., if the approximation turns out to be too coarse, the approximation is automatically refined up to a point for which it either generates a counter example or disprove it. While this technique is adaptive, our method is not. Therefore our approach can be seen as an instance of a first time right for CEGAR.

D-Finder [BBNS09] presents a compositional method for checking invariant properties. The basis of the method is an algorithm that by iterations computes invariants for components until they are strong enough to imply a global invariant that needs to be checked. In contrast to our method, where an over-approximation of the model is obtained, D-finder over-approximates the local properties of the component.

Another approach related to ours, can be found in VeriSoft [God97]. Their approach consists of a systematic exploration of a state space by executing arbitrary code written in any language. They guarantee complete state space coverage up to some depth, hence a partial state space exploration. Consequently, this only guarantees safety properties up to a certain depth/bound and not for the entire system.

A last piece of related work can be found in program slicing [Wei81]. This technique selects parts of the source code that are of interest to the values of specific variables. Our approach takes this to the extreme by abstracting from all the variables and focusing on calls between interfaces. Perhaps the technique of program slicing could also have been made instrumental in abstracting from less relevant aspects of the model such as the logging of events.

4.8 Conclusions

This chapter shows how safety properties are verified for complex systems. With the help of an intermediate programming language, we have proposed a procedure that transforms code into an mCRL2 model, thereby preserving a simulation relation. The accompanied case study demonstrates that such an approach is applicable to verify safety requirements in an industrial setting.

Similar methods are used in combination with other implementation languages (C, C++, C#, Pascal, Delphi, Java ...) and verification languages. Mentionable, they

have similar constructs for describing behavior such as synchronized communication, sorts to encode different processes and non-determinism.

Although it is possible to verify safety requirements, it still requires an engineer to validate the results on the actual model. If data would have been included, we would not have to perform these steps by hand. For small models it might be possible to include these relevant conditions. However, including them in industrial systems renders any verification run useless. Hence, compromises have to be made.

The proposed method, that is illustrated here, is not beneficial for requirements other than the safety ones. Since we reason over an over-approximation it could be that a liveness property holds for the model, but it does not hold for the implemented system. To verify them, the model needs to be enriched with the elementary required data. However, enriching the model is again a labor intensive task.

The process of creating an ad-hoc formal model in a similar way is labor intensive, because an engineer needs to apply a suitable abstraction, often performs the transformation by hand, and needs to inspect the validity of the transformed model by conducting tests. This makes the task challenging and also in a sense somewhat unpredictable. A wrong abstraction could lead to a model for which the verification becomes impossible (e.g., state space explosion) or too trivial (e.g., all the requirements appear to hold). This means that automating this route, i.e., the transformation of code to a formal model, is difficult to perform without the help of an engineer. The derivation and the implementation of the entire model took approximately three months.

Furthermore, we would like to address that the model has been extracted from the code of a stable software branch, where no further development was performed. Therefore we did not have to take the synchronization of the models with updated code into account. If one would apply this process during development, one should consider this challenge as well.

Based on the amount of related work that is presented here, and can be found in literature the methods that try to create formal models from code are versatile. For all of the techniques we see that either only small models can be formally checked if no, or hardly any, abstractions are applied. To model and verify larger and more complex system more rigorous abstraction and ad-hoc techniques are required. In turn, these methods require more ingenuity and are harder to implement in an automated translation. Hence, they are less suitable to be applied in an industrial setting.

Chapter 5

Modeling Specification Languages

5.1 Introduction

A successful system development requires the cooperation of different disciplines. Traditionally, every discipline is concerned with a separate development trajectory, for which the various disciplines use their own methods, techniques and tools to construct their behavioral models. As the different developments are often performed in isolation and are merged near the end of the development trajectories, it can result in difficulties during system integration.

To overcome these problems, attempts have been made do define a single language that spans over different disciplines to facilitate a multidisciplinary modeling environment. Such a language enables the study of different aspects in isolation, while maintaining the consistency for a global model. A language that tries to achieve this is the Chi 2.0 language [BHR+08]. This language along with its tools, is a formalism suitable to *model* and *simulate* hybrid systems. A hybrid system combines discrete events with continuous behavior. The language integrates concepts from dynamics, control theory and computer science. The language has evolved from the work of (Hybrid) χ [BMR+06], for which the roots are found in CSP [Hoa78] and hybrid automata [Hen96].

The Chi 2.0 language targets the study of performance parameters (e.g., throughput, cycle time, system occupation) for hybrid systems by means of *simulation* techniques. Though, the semantics is formally defined and is supported by tooling, it does not offer any means to *verify* behavioral properties. To resolve this limitation, we explore an ad-hoc method for deriving formal behavioral models: defining a denotational relation that allows for a compositional transformation of Chi 2.0 specifications into mCRL2 specifications. This chapter describes an instance of a bridge between the interchange environment and the analysis environment from Figure 1.1.

Because the Chi 2.0 language is a hybrid language, it incorporates aspects that are

hard or even impossible to translate. Hence, we only translate those Chi 2.0 notions that comply to a specific format. We first describe the design decisions for the input format as well as the informal rationale to overcome the discrepancies between the languages. These descriptions are guided by the operational semantics of the Chi 2.0 language [BHR+08], and the operational semantics of the mCRL2 language, including its timed fragment (Chapter 2.2.3). Although the discrete parts of the languages are considered to be relatively close to each other, we see that the transformation is non-trivial due to the differences between the languages and by restrictions imposed by the tools.

The outline of this chapter is as follows. In Chapter 5.2 we introduce the Chi 2.0 language, describe the syntax and semantics of the Chi 2.0 language, and provide the design decisions that we respect during the transformation. Chapter 5.3 provides the compositional transformation scheme per syntactic Chi 2.0 notion. In Chapter 5.4 we state additional considerations that allow for more models to be translated. Chapter 5.5 validates the transformation by evaluating four translated models. Chapter 5.6 describes related work. Chapter 5.7 concludes this chapter.

5.2 Syntax and Semantics of the Chi 2.0 language

The Chi 2.0 language is a hybrid modeling formalism that integrates concepts from dynamics and control theory with concepts of process algebra and hybrid automata. The language consists of five important language concepts. The first concept concerns the *different variable classes*. Variables are either discrete, continuous or algebraic. The second concept is the *strong time determinism, combined with delayable guards*. The third concept is the use of *urgent and non-urgent actions*. The fourth concept denotes the *algebraic (ordinary differential) equations*. The last concept specifies the *different interaction mechanisms* between concurrent processes by one of the following three methods. The first interaction method provides a handshake synchronization, that allows for the synchronous communication between discrete event processes. The second interaction method facilitates the sharing of variables between concurrent hybrid processes. The third interaction method specifies the synchronization through shared action labels, i.e., a synchronizing action is only executed when for all of the alphabets of the concurrent processes in which the action-labels occur, all of the relating actions are simultaneously enabled.

The discrete part of the language allows notions to specify complex system behavior. It provides process definitions and process references, which enable process re-use, encapsulation and hierarchical composition of processes. The language also offers process terms for scoping, local variables, local channels, recursion and channel hiding.

5.2.1 Syntactic and Semantic Differences

The Chi 2.0 language and the mCRL2 language have been developed for different engineering disciplines. Therefore not all notations of the Chi 2.0 language are denotable in the mCRL2 language. This section, provides the important syntactic and semantic differences between the languages. It also states the design decisions to resolve differences between the languages. Some decisions require non-trivial modeling decisions. The implementations of these decisions are provided in Chapter 5.3.

Kind of Languages

The unrestricted Chi 2.0 language allows the specification of hybrid processes. The LTSs that correspond to the semantics are expressed by three transition relations, namely action transitions, continuous flow transitions and consistency transitions. The semantics that corresponds to the behavior of an mCRL2 specification is described by a timed-LTS, relating the behavior of discrete events to transitions at a certain moment in time. The progression of time is represented by an idle relation (Chapter 2.2.3). Based on these observations there are three difficulties for which we provide design decisions:

Decision 5.1: When translating a Chi 2.0 specification, we only consider its timed discrete event behavior. The tools that currently accompany the mCRL2 language do not facilitate any means to solve differential equations. Solving these equations requires complex (symbolic) transformations to calculate their exact solutions. As it is not guaranteed that exact solutions are present, and approximations are imprecise, we ignore the differential equations as they can render the outcome of a verification possibly doubtful. Therefore, we restrict the continuous model variables to only the special variable time.

Algebraic model variables can be interpreted as variables that are defined through a function. Since we eliminate continuous variables, the algebraic variables can be substituted by their corresponding functions. For the translation we assume that they are. Hence, the set of algebraic variables is presumed empty. ⌟

Decision 5.2: The Chi 2.0 language can specify consistent and inconsistent behavior. A consistent process may allow the progress of time, perform an action or deadlock. An inconsistent process is prohibited from doing so. Consistency in the Chi 2.0 language is specified through e.g., invariants. The mCRL2 language has no notion for expressing inconsistency. Hence, we assume that all Chi 2.0 processes are consistent. Therefore we do not allow any process terms that introduce inconsistent behavior. ⌟

Decision 5.3: The semantics of the Chi 2.0 language has different transition relations. To model the behavior in the mCRL2 language we map the transition relation to the timed transition relation. The continuous behavior of the Chi 2.0 language is mapped to the idle relation in the mCRL2 language. ⌟

Model Variables

The Chi 2.0 language has model variables and assignments. The mCRL2 language has no concept of model variables and assignments to those, since the language is stateless. Based on this observation we take the following design decision:

Decision 5.4: To capture model variables, we introduce a memory process in the mCRL2 specification that facilitates the (global) variable management. For obtaining and altering the values of variables (e.g., as a consequence of an assignment), the concepts are modeled by enforcing a value exchange for only the relevant model variables between the memory process and the translated process. For every variable we introduce a separate set of actions that exchange values between the memory process and the translated specification. ⌟

Action Labels

The action transition of a Chi 2.0 process shows the (restricted) quadruple "σ, l, W, σ'". The mCRL2 language can only perform multi-actions.

Decision 5.5: To capture the quadruple in the translated mCRL2 specification, we introduce for each of the quadruple elements a separate representation. The individual representations are merged into a single mCRL2 multi-action. ⌟

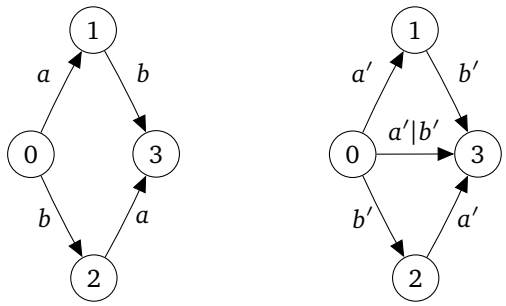
Decision 5.6: We presume that for every action that is specified in a Chi 2.0 specification, we have a corresponding action declared in an mCRL2 specification. ⌟

Parallel Composition

The parallel composition of the Chi 2.0 language differs from the parallel composition of the mCRL2 language. The Chi 2.0 language strictly interleaves the actions from the concurrent processes. Corresponding send and receive actions are the only synchronizing actions. In the mCRL2 language actions are interleaved differently, i.e., the strict interleaving is extended with the synchronized execution of actions. The effect is described by a multi-action, where multiple actions are performed simultaneously.

Example 5.1(Concurrent execution). This example specifies the Chi 2.0 process $a \parallel b$ and the mCRL2 process $a' \parallel b'$. Figure 5.1(a) depicts the actions that can be executed by the Chi 2.0 process. Figure 5.1(b) depicts the actions that can be executed by the mCRL2 process. By allowing the synchronized execution of actions, we see that Figure 5.1(b) depicts an additional transition that is not shown in Figure 5.1(a). △

Decision 5.7: The additional multi-actions that occur during the execution of an mCRL2 process, pose additional actions that cannot be performed by a Chi 2.0 specification. Hence, if we translate a Chi 2.0 parallel composition by an mCRL2 parallel composition, we need to restrict the translated process by the allow operator. In this



(a) The Chi 2.0 process $a \parallel b$ (b) The mCRL2 process $a' \parallel b'$

way we exactly specify the allowed actions of the interleaving process. ⌟

If we interpret an internal Chi 2.0 action (τ) as an mCRL2 internal action (τ), it is possible that a Chi 2.0 internal action is performed simultaneously with another action in the translated process. Any mCRL2 multi-action that is extended with an internal action τ resides in the same multi-action equivalence class.

Decision 5.8: To prevent such behavior, all τ actions of a Chi 2.0 specification are represented in the translated mCRL2 process by the action τ_χ. The action τ_χ is then hidden at the outermost level. ⌟

Different Models of Time

Time is treated differently by both languages. It differs in two aspects. Firstly, time in the Chi 2.0 language is relative, meaning that the passage of time is measured from a previous action. Time within the mCRL2 language is absolute, meaning that all timings refer to a (single) global clock [BB97]. Secondly, the Chi 2.0 language allows the sequential composition of two (or more) actions at the same moment in time, while preserving their mutual order. Compared to the semantics of the mCRL2 language, the processes here behave differently. mCRL2 actions are (i) forced to happen either simultaneously at the same moment in time thereby *losing* their mutual order, or (ii) need to be performed at different time instances, thereby *preserving* their mutual order. Based on these observations, we take the following three design decisions:

Decision 5.9: Since the mCRL2 language is stateless, we introduce an extra mCRL2 process that stores the cumulative time value, i.e., it stores the time value that has elapsed since the start of a model. ⌟

Decision 5.10: The time inconsistency is bridged by defining a time domain with micro steps in the mCRL2 language. The new time value consists of a real number representing the absolute time value, and a counter denoting the nth action performed at a certain moment in time. ⌟

Decision 5.11: For both formalisms it is possible to model deadlocks at a certain moment in time. Decision 5.9 states that the time value is stored in a separate process. If we represent a deadlock by simply modeling a δ in the mCRL2 specification, we are unable to retrieve the current time value. The functionality is required when we model the time can progress operator (See Chapter 5.3.6). So, we always exchange the time value of the time process with the translated process to retrieve its current time value.

⌟

Alternative Composition

The alternative composition of the Chi 2.0 language behaves strongly time deterministic [BR04]. The alternative composition of the mCRL2 language behaves weakly time deterministic. In a strong time deterministic formalism it is not possible that time may determine any choice. In a weak time deterministic formalism time may progress, thereby disabling choices. The difference between the languages is illustrated in the following example.

Example 5.2(Weak and Strong Time Determinism). If we have the choice to perform an action a at time 3 and an action b at time 5, we can only perform the action a at time 3 in a strong time deterministic setting. If the same choice is proposed in a weak time deterministic setting, both alternatives are considered. △

To overcome the difference we take the following design decision:

Decision 5.12: The Chi 2.0 alternative composition ([]) is modeled by the mCRL2 alternative composition (+). As the mCRL2 operator allows more behavior, we constrain the deterministic choice, by introducing a special function Δ_{time}^{max} that calculates the ultimate delay for the process terms on both sides of an alternative composition operator. The function takes the lowest time upper-bound for which both alternatives can delay. The outcome of the function denotes a time value and is added as a constraint that denotes the maximal time under which the mCRL2 choice is performed. Thus the strong deterministic choice is mimicked in a weak time deterministic setting. ⌟

Scopes

The syntax of the Chi 2.0 language defines local variables, local recursion scopes, local actions, and local communications. The mCRL2 language is stateless, so it has no (local) model variables. Furthermore, recursion and action declarations are defined globally. Local communication within the mCRL2 language is performed by applying the communication operator to a process. The scopes are modeled by taking separate design decisions for each of them.

Decision 5.13: We assume that all model variables have unique names. In a Chi 2.0 process it is possible to define local model variables. These variables can be accessible

at a local level but are hidden for the surrounding processes. To avoid that variables are bound to multiple values, the semantics of the Chi 2.0 language states that during the execution of a local variable scope, the local variables are replaced by freshly chosen variables. When we assume that all local variables have globally unique names, then we do not have to substitute them by freshly chosen ones.

The mCRL2 language has no notion of local variables. Because we assume that all local variables definitions are globally unique, we encapsulate in the mCRL2 model the non-successful value exchanges based on their variable names thereby preventing that they are accessed by surrounding processes. Furthermore we hide the values of the local variables, as they are not visible at the outermost level. Because non-successful value exchanges are blocked, always the most inner-most local variables are exchanged. The functionality is required when variable scopes are nested in recursion scopes.

The uniquification of the Chi 2.0 model variables is a syntactic pre-processing step on the Chi 2.0 specification. ⌟

Decision 5.14: We assume that all recursion scope variables have unique names. The Chi 2.0 language defines recursion scopes locally. To avoid that they are defined multiple times, the semantics of the Chi 2.0 language states that during the execution of a recursion scope, the locally introduced modes are replaced by freshly chosen ones.

In the mCRL2 language, the recursive processes are globally defined. We assume that all definitions of local modes are globally unique in the Chi 2.0 specification. So, it is not required to substitute them by freshly chosen ones. This also implies that all recursion scopes can be directly translated into process definitions in the mCRL2 language. Hence, we introduce for every recursion mapping in the recursion scope a separate mCRL2 process equation.

The uniquification of the Chi 2.0 modes is a syntactic pre-processing step on the Chi 2.0 specification. ⌟

Decision 5.15: We assume that all action declarations have globally unique names. The Chi 2.0 language defines actions locally. That means that these actions are available inside the scope, but are inaccessible (hidden) outside. We have assumed that all Chi 2.0 actions are modeled by mCRL2 actions (Decision 5.6). The action declarations for these mCRL2 actions are globally defined. While all locally declared Chi 2.0 actions are globally unique, there is no need to substitute them. Under this assumption we model the local actions by corresponding mCRL2 actions.

To model the abstractions on the local actions, we rename these actions to τ_χ, thereby taking Decision 5.8 into account.

The uniquification of the Chi 2.0 local action labels is a syntactic pre-processing step on the Chi 2.0 specification. ⌟

Decision 5.16: We assume that all channels have globally unique names. Like local action scopes, channels are defined locally. The successful communications are hidden outside the scope and non-successful communications are blocked. Communication

labels are defined locally. They are replaced by freshly chosen communication labels during execution.

The communication actions are modeled by mCRL2 actions. So, the mCRL2 actions need to be declared globally. As we have seen with local action labels, we assume that every locally introduced Chi 2.0 communication label is also globally unique. This assumption asserts that no substitutions are required. Hence, the actions that locally describe the communication can be declared globally and are therefore unique.

Outside the scope, the successful communication actions are hidden, the non-successful communication actions are blocked. To hide the actions we take Decision 5.8 into account. The blocking is modeled by the mCRL2 encapsulation operator.

The uniquification of the Chi 2.0 communication labels is a syntactic pre-processing step on the Chi 2.0 specification. ⌟

Simplification

To restrict the complexity of the transformation, we apply simplifications to the some of the Chi 2.0 concepts. This results in the following design decisions.

Decision 5.17: The syntax of the Chi 2.0 language allows a vector of arbitrary, but finite length to communicate values over channels. We assume that communications only communicate a single value. ⌟

Decision 5.18: An urgency mapping in a Chi 2.0 specification dictates whether actions and *successful* communications are performed urgent or non-urgent. We assume that actions are can be both.

We assume that channels are non-urgent, because of the following reason. Successful communications originate from communicating channel ends. Times at which the actions occur are independently determined and guards can only reason on data (therefore not over actions). If we assume that a communication is urgent, then it is difficult to compute the first moment in time at which a successful communication is performed. If process terms are encoded into data expressions, it would be possible to do so, however it would complicate the transformation dramatically.

In Chapter 5.4.4 we present and motive an alternative solution that partly resolves the urgency for channels. The solution performs a slight (but acceptable) change to the semantics of the Chi 2.0 language in order to deal with urgency.

⌟

Decision 5.19: A valuation in the Chi 2.0 language relates variables to values, which may be undefined, i.e., $\perp$. For presentation purposes, we assume that when a variable occurs in the valuation it always corresponds to a concrete value. This assumption ensures that variable to value mappings are never undefined. ⌟

5.2.2 Syntax

This section presents the restricted syntax and semantics of the Chi 2.0 language. A full description of the language is provided in [BHR+08].

Notions

The set of all values is denoted by Λ. The set of all variables is denoted by $\mathcal{V}$, which includes the reserved variable time. $\Sigma = \mathcal{V} \mapsto \Lambda$ denotes the set of all variable valuations. A variable valuation is a partial function from variables to values, which captures the values of variables at a certain moment in time.

The set of basic action labels is $\mathcal{L}_{\text{basic}}$. Provided that $\mathcal{H}$ denotes the set of communication channels, then the set of all communication labels $\mathcal{L}_{\text{comm}}$ is defined by $\{h!cs, h?cs \mid h \in \mathcal{H}, cs \in \Lambda\}$. The set of all action labels $\mathcal{L}$, excluding τ, is defined as $\mathcal{L} = \mathcal{L}_{\text{comm}} \cup \mathcal{L}_{\text{basic}}$, assuming $\mathcal{L}_{\text{basic}} \cap \mathcal{L}_{\text{comm}} = \emptyset$. The set of action labels $\mathcal{L}_t$, including the internal action τ, is specified as $\mathcal{L}_t = \mathcal{L} \cup \{\tau\}$. The set of all urgency mappings for action labels and communication labels, including τ, is denoted by the partial function $\mathcal{U}_{\text{ah}} = (\mathcal{L}_{\text{basic}} \cup \{\tau\} \cup \mathcal{H}) \mapsto \mathbb{B}$.

The sort $\mathcal{T} = \mathbb{R}$ denotes the set of all time points. The variable time expresses the amount of time that has progressed since a Chi 2.0 process has started.

The set of recursion variables (modes) is denoted by $\mathcal{M}$. The sort P_{proc} defines the set of the process terms. Sort $\mathcal{R} = \mathcal{M} \mapsto P_{\text{proc}}$ denotes the recursion mappings as a partial function from recursion variables to process terms.

The dynamic variable mapping $\mathcal{D} = \mathcal{V} \mapsto \{\mathsf{disc}, \mathsf{cont}\}$ denotes the set of all dynamic type mappings, i.e., partial functions from variables to the dynamic types $\{\mathsf{disc}, \mathsf{cont}\}$. Here, disc denotes the dynamic type for discrete variables and cont denotes the dynamic type for continuous variables. Let $D \in \mathcal{D}$ be a dynamic variable mapping then D is defined as:

$$\begin{cases} \mathsf{time} \in dom(D) \Rightarrow (D(\mathsf{time}) = \mathsf{cont}) \\ \forall_{v \in dom(D) \setminus \{\mathsf{time}\}} (D(v) = \mathsf{disc}) \end{cases}$$

D_{disc} and D_{cont} denote the sets of discrete and continuous variables respectively, i.e., defined as $D_t = \{x \in dom(D) \mid D(x) = t\}$ for $t \in \{\mathsf{disc}, \mathsf{cont}\}$.

Abstract Syntax

The relevant (abstract) syntax for the Chi 2.0 language is given by the following grammar:

$$\begin{array}{lcl} P_{\text{atom}} & ::= & \mathsf{tcp}\, u \mid u \rightarrow a : W : r \mid u \rightarrow h\,!\,e : W : r \mid u \rightarrow h\,?\,x : W : r \mid X \\ P_{\text{proc}} & ::= & P_{\text{atom}} \mid P_{\text{proc}};P_{\text{proc}} \mid P_{\text{proc}} \,[\!]\, P_{\text{proc}} \mid P_{\text{proc}} \parallel P_{\text{proc}} \mid \partial_H(P_{\text{proc}}) \mid \\ & & |[_{\text{R}}\, R :: P_{\text{proc}}\,]| \mid |[_{\text{A}}\, U_A :: P_{\text{proc}}\,]| \mid |[_{\text{H}}\, U_H :: P_{\text{proc}}\,]| \mid |[_{\text{V}}\, D, \sigma :: P_{\text{proc}}\,]| \end{array}$$

Atomic terms P_{atom} defines the atomic terms, where u is a predicate over model variables, a is an action label from the set of basic action labels $\mathcal{L}_{\text{basic}}$ or the internal action τ ($a \in \mathcal{L}_{\text{basic}} \cup \{\tau\}$). The set of variables that are allowed to change their values during execution is indicated by W. The update predicate r expresses the way in which the values of the model variables change. A channel name is denoted by h, e is an expression, and x is a model variable. Finally, X is a recursion variable from $\mathcal{M}$.

Predicates are arbitrary Boolean expressions containing model variables. Update predicates are arbitrary expressions containing $-$-super-scripted and ordinary occurrences of model variables. For example, the predicate $x + y = x^- + y^-$ describes that the sum of the values of x and y remains the same to the sum of x^- and y^-.

The time can progress process term $\mathsf{tcp}\, u$ restricts the progress of time. It specifies local urgency, by allowing delays as long as predicate u is satisfied.

The atomic process term $u \rightarrow a : W : r$ is called a guarded action. Here, u describes a condition on the model variables for which the action a is allowed to occur. The set W describes the model variables that are allowed to be updated by the execution of a. The variable r describes the update predicate.

The atomic process terms $u \rightarrow h\,!\,e : W : r$ and $u \rightarrow h\,?\,x : W : r$ denote respectively the guarded send and the guarded receive process term. The intuitive meaning of $u \rightarrow h\,!\,e : W : r$ is that the value of expression e is sent over channel h provided that predicate u holds. The model variables from the set W are allowed to change according to the update predicate r. The meaning of $u \rightarrow h\,?\,x : W : r$ is that, providing that predicate u holds, a value is received via channel h and the value is stored in model variable x. Furthermore, the model variables from set $W \cup \{x\}$ are allowed to change according to the update predicate r.

Process terms The process terms are defined by P_{proc}. The sequential composition is denoted by $P_{\text{proc}};P_{\text{proc}}$. The (non-deterministic) choice between alternatives is represented by $P_{\text{proc}} \,[\!]\, P_{\text{proc}}$. The parallel composition is defined by $P_{\text{proc}} \parallel P_{\text{proc}}$. Channel encapsulation is shown in $\partial_{\mathcal{H}}(P_{\text{proc}})$, where $\mathcal{H}$ denotes a set of channels that are encapsulated (blocked). By means of the recursion scope operator $|[_{\text{R}}\, R :: P_{\text{proc}}\,]|$, local recursion definitions are introduced in a Chi 2.0 process, where $R \in \mathcal{R}$ denotes a recursion mapping. With the help of an action scope operator $|[_{\text{A}}\, U_A :: P_{\text{proc}}\,]|$, where $U_A \in \mathcal{U}_{ah}$, $dom(U_A) \subseteq \mathcal{L}$, denotes the urgency mapping for action labels, local actions are visible within the scope, but are hidden outside the scope operator. The process $|[_{\text{H}}\, U_H :: P_{\text{proc}}\,]|$ describes local channels, with $U_H \in \mathcal{U}_{ah}$, $dom(U_H) \subseteq \mathcal{H}$, as the urgency mapping for channels. Communication actions on local channels are hidden outside the scope operator. The process $|[_{\text{V}}\, D, \sigma :: P_{\text{proc}}\,]|$ defines a variable scope and defines local variables, where $D \in \mathcal{D}$ specifies a dynamic type mapping, and σ a valuation.

Omitted Abstract Syntax

The remainder of this section describes the omitted concepts from the grammar and their rationales.

Equations and Invariants Equations and invariants describe the evolution of continuous and algebraic variables over time. Because of Decision 5.1 the syntax for equations and the syntax for invariants have become irrelevant, and are omitted.

Concrete Syntax The concrete syntactical notations (i.e., abbreviations) that are offered by the Chi 2.0 language are not included in the transformation. The abbreviations provide a comfortable syntax to express e.g., deadlock, multi-assignments and inconsistency. The comfortable syntax is denotationally defined by the abstract notions that are covered in the translation. Furthermore, we chose not to transform the guarded communication update $(h!?x{:=}e : W : r)$, since the abbreviation only eliminates communicating channels in a parallel composition.

Model Inconsistency Decision 5.2 states that a Chi 2.0 process cannot get into an inconsistent state by algebraic or continuous variables. The only place that can introduce an inconsistent state is the initialization operator. Hence, we omit the initialization operator from the set of translatable concepts.

Synchronizing Actions The behavior of synchronizing actions cannot be mimicked by an mCRL2 specification. The synchronizing actions in a Chi 2.0 specification allow the synchronous execution of all actions that carry the same label. The synchronization can potentially range over infinitely many actions. Therefore it is not possible to capture the intended behavior with the help of an mCRL2's allow or an mCRL2's encapsulation operator. Hence, we assume that (i) all action labels are non-synchronizing, and (ii) a Chi 2.0 specification does not contain an action synchronization operator.

5.2.3 Semantics

The semantics is described through SOS, provided with a context of environment variables. Here, we present the context restricted to the relevant parts. The non-restricted deduction rules are found in [BHR+08]. These non-restricted rules are omitted here, but have inspired us to define the transformation. We only sketch the associated behavior and translate that into corresponding mCRL2 notions. The relation between the semantics of the languages is explained in Appendix A.1.

The semantics of the restricted Chi 2.0 language, associates a Chi 2.0 process $\langle p, \sigma, E\rangle$ to an LTS that describes action transitions and continuous behavior. Because of Decision 5.2 we do not consider the consistency transitions. Here, $p \in P_{\text{proc}}$ denotes a process term, σ denotes a valuation, and E denotes an environment. The environment E is in itself a quadruple (D, U, J, R), where $D \in \mathcal{D}$ describes a dynamic type mapping, $U \in \mathcal{U}_{\text{ah}}$ defines an urgency mapping, $J \subseteq \mathcal{V}$ denotes the set of jumping variables, and $R \in \mathcal{R}$ denotes the recursions scopes. We define $\mathcal{E} = \mathcal{D} \times \mathcal{U}_{\text{ah}} \times 2^{\mathcal{V}} \times \mathcal{R}$.

Decision 5.1 restricts the semantics in a similar way as the work of [BMR+05]. There, the authors restrict the hybrid χ language [BMR+06] to a timed setting. The (reduced) relations that we consider after the restriction are:

- **(Terminating) Action transitions**

  $\_ \xrightarrow{-} \_ \subseteq (P_{\text{proc}} \times \Sigma \times \mathcal{E}) \times (\Sigma \times \mathcal{L}_t \times 2^{\mathcal{V}} \times \Sigma) \times (P_{\text{proc}} \cup \{\surd\} \times \Sigma \times \mathcal{E})$

The intuition of a (termination) action transition $\langle p, \sigma, E\rangle \xrightarrow{\sigma, l, W, \sigma'} \langle p', \sigma', E'\rangle$ is that a process $\langle p, \sigma, E\rangle$ executes a discrete action $l \in \mathcal{L}_t$ with visible valuations σ and σ' and W represents the set of externally visible discrete variables that are allowed to change (jump) during an action transition, and transforms into the process $\langle p', \sigma', E'\rangle$. Here, σ' and E' respectively denote the valuation and environment of the process term p' after the discrete action l is executed. If p' equals $\checkmark$, then the action transition describes a terminating action transition.

- **Continuous behavior**

  $\_ \overset{-}{\longmapsto} \_ \subseteq (P_{\text{proc}} \times \Sigma \times \mathcal{E}) \times (\mathcal{T}_{\geq 0} \times \Sigma \times (\mathcal{T} \mapsto val_{\text{ah}})^3) \times (P_{\text{proc}} \times \Sigma \times \mathcal{E})$ where the set of all action/channel valuations are defined by $val_{\text{ah}} = (\mathcal{L}_{\text{basic}} \cup \{\tau\} \cup \mathcal{H}) \mapsto \mathbb{B}$ which describes a mapping from action labels and channels to a Boolean value, representing the value of a guard associated to the action label or channel. The restricted continuous behavior only models the progression of time $\langle p, \sigma, E\rangle \overset{t, \sigma, (\theta_n, \theta_s, \theta_r)}{\longmapsto} \langle p', \sigma', E'\rangle$. During the time transition the valuation of the visible variables remain constant, which is specified by σ. At the end-point t, the process results in $\langle p', \sigma', E'\rangle$. The triple $(\theta_n, \theta_s, \theta_r)$ represents three trajectories that describe the guards during the delay ($s \in [0, t]$) for the associated action and channel labels. The first trajectory θ_n represents the guard trajectory for the non-synchronizing action labels and the communicated channels. The second trajectory θ_s and third trajectory θ_r respectively represent the guard trajectories for the non-communicated send action and receive action channels.

5.3 Translation Scheme

The relevant syntactic notions are transformed from the Chi 2.0 language to the mCRL2 language. We first introduce the time sort with micro steps. Then we relate the transition relations between the different formalisms. Hereafter we describe how the urgency mappings are modeled in the mCRL2 specification. Subsequently, we provide the translation function, that includes the translation functions for the model variables, the environment variables and the accumulated time. The details of the translation for the atomic terms and the different process terms, are found in the concluding part of this section.

Interpretation

The transformation uses an interpretation function $[\![\cdot]\!]$, which takes a syntactic notion from the Chi 2.0 language and expresses the syntactic counterpart notion in the mCRL2 language. The interpretation function expresses obvious interpretations, e.g., if S is a sort in a Chi 2.0 specification then $[\![S]\!]$ expresses the mCRL2 representation of that sort. Likewise, if v is a variable in a Chi 2.0 specification, then $[\![v]\!]$ expresses the mCRL2 representation of that variable.

5.3.1 Time with Micro Steps

Based on Decision 5.10 we model a time domain with micro steps, expressed by the mCRL2 structured sort $Time_{\mathrm{H}}$. The sort $Time_{\mathrm{H}}$ consists of a tuple, where the first element represents the time value and the second element represents a counter, counting the nth action for that specific time value. The type of the first element is $\mathbb{R}$ (reals) and the type of the second element is $\mathbb{N}$ (natural numbers). We assume that the time value is non-negative.

sort $\quad Time_{\mathrm{H}} = \textbf{struct}\ time_{\mathrm{H}}(\pi_{time} : \mathbb{R}, \pi_{counter} : \mathbb{N});$

5.3.2 Ultimate Delay Function

Decision 5.12 requires a function that computes the ultimate delay. The function is represented by $\Delta^{max}_{time} : P_{\mathrm{proc}} \times \vec{\Lambda} \times \mathbb{R} \rightarrow \mathbb{R}$ that requires three arguments. Let $p, \vec{v}, t$ be these arguments. The ultimate delay function computes the maximal amount of time for which a process $p \in P_{\mathrm{proc}}$, i.e., the first argument, may delay its execution. The second argument $\vec{v}$ specifies a vector with the current values of *all* the discrete model variables. The values for model variables that are irrelevant may be set to arbitrary values. The third argument denotes the current time value t.

5.3.3 Relating Transition Relations

Decision 5.5 relates a Chi 2.0 action (σ, l, W, σ') to an mCRL2 multi-action. The quadruple is related to a multi-action in the following form:

- the valuation σ is associated to the set of actions $com^{[\![v]\!]}_{\mathrm{mem}} : \Lambda$, such that for every variable $v \in D_{\mathsf{disc}}$ an action $com^{[\![v]\!]}_{\mathrm{mem}}$ is used with the value of that variable in the valuation as a data parameter, i.e., the action $com^{[\![v]\!]}_{\mathrm{mem}}([\![\sigma(v)]\!])$ represents that variable v has value $\sigma(v)$. For every discrete variable v that syntactically occurs in the Chi 2.0 specification we assume that the mCRL2 specification contains the following set of action declarations:

  act $\quad com^{[\![v_1]\!]}_{\mathrm{mem}}, \ldots, com^{[\![v_n]\!]}_{\mathrm{mem}} : \Lambda;$

- The argument time from valuation σ', i.e., $\sigma(\mathsf{time})$ is modeled by the action $com_{\mathrm{time}} : Time_{\mathrm{H}}$. Note that if $t \in Time_{\mathrm{H}}$, then the mCRL2 data expression $\pi_{time}(t)$ corresponds to the Chi 2.0 value for time.

- the Chi 2.0 action label $l \in Act_{Lab}$ is modeled by an mCRL2 action l. For every action label l that syntactically occurs in the Chi 2.0 specification we assume an action declaration:

  act $\quad l;$

If we write $[\![a]\!]$ and a is Chi 2.0 action, then it is translated to the mCRL2 action a. An internal Chi 2.0, i.e., $[\![\tau]\!]$ is translated by a τ_χ labeled mCRL2 action. Hence we introduce:

act τ_χ;

For every communication channel h that syntactically occurs in the Chi 2.0 specification we assume that the following actions are declared:

act $send_{[\![h]\!]}, recv_{[\![h]\!]}, comm_{[\![h]\!]} : \Lambda$;

- As it is not allowed to directly define data as mCRL2 actions, we introduce the action *diff* that captures the set of changing variables in the action parameter. This implicitly assumes that the mCRL2 specification contains the following action declaration:

  act $diff : Set([\![\mathcal{V}]\!])$;

  where sort $[\![\mathcal{V}]\!]$ specifies the set of all interpreted Chi 2.0 variables. Communications where the same sets of variables are allowed to change on both the sending and the receiving side are modeled by single *diff*-action in a multi-action. Communications where different sets of variables are allowed to change on the sending and the receiving side are modeled by two *diff*-actions in the multi-action.

- Analogue to modeling σ, the valuation σ is associated to the set of actions $com^{[\![v]\!]}_{\text{mem}'} : \Lambda$, such that for every variable $v \in dom(\sigma') \setminus \{\mathsf{time}\}$ an action $com^{[\![v]\!]}_{\text{mem}'}$ is used with the updated value of that variable in the valuation as a data parameter, i.e., the action $com^{[\![v]\!]}_{\text{mem}'}([\![\sigma'(v)]\!])$ represents that variable v has the updated value $\sigma(v)$. For every variable v that syntactically occurs in the Chi 2.0 specification we assume that the mCRL2 specification contains the following set of action declarations:

  act $com^{[\![v_1]\!]}_{\text{mem}'}, \ldots, com^{[\![v_n]\!]}_{\text{mem}'} : \Lambda$;

- The argument time from valuation σ', i.e., $\sigma'(\mathsf{time})$, is modeled by the action $com_{\text{time}'} : Time_{\text{H}}$.

- We furthermore assume that all action declarations are mutually disjoint.

5.3.4 Global Urgency Mapping

Decision 5.15 and Decision 5.16 assume that all actions and communications are globally unique. Hence we define a global mapping in the mCRL2 specification that specifies for an action its corresponding urgency. Let U_G be an urgency mapping

that defines all the urgency mappings for the locally and globally defined actions and communications in a Chi 2.0 specification. If $U_G = \{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$, we model the urgency mapping in the mCRL2 specification as:

$$\begin{array}{ll} \textbf{map} & U_G : [[\mathcal{L}_t]] \to \mathbb{B}; \\ \textbf{eqn} & U_G(a_1) = b_1; \\ & \quad\vdots \\ & U_G(a_n) = b_n; \end{array}$$

Note that all actions that appear in a Chi 2.0 specification occur exactly once in the urgency mappings. We assume that the sort $[[\mathcal{L}_t]]$ in the above mCRL2 specification, is the counterpart representation for $\mathcal{L}_t$ that denotes the occurring labels in the Chi 2.0 specification. Resulting from Decision 5.18 we assume that all channels are non-urgent:

$$\forall_{h \in \mathcal{H}} U_G([[h?]]) = \mathit{false} \wedge U_G([[h!]]) = \mathit{false}$$

5.3.5 Translating a Specification

Let $\langle p, \sigma, (D, U, J, R)\rangle$ be a Chi 2.0 model that consists of a process term p, an initial (global) valuation σ and an environment (D, U, J, R). Then with the help of function $\mathcal{F}_{\text{init}} : P_{\text{proc}} \times \Sigma \times \mathcal{E} \to P_{\text{mCRL2}}$ we compute the transformation for the model $\langle p, \sigma, (D, U, J, R)\rangle$ where P_{mCRL2} denotes the resulting mCRL2 model.

The translation consists of the concatenation of four (partial) mCRL2 specifications, obtained by different functions. The first function $\mathcal{F}_{\text{Mem}} : 2^{\mathcal{V}} \to P_{\text{mCRL2}}$ models the memory process that is associated to the set of discrete variables derived from the initial valuation σ. The second function $\mathcal{F}_{\text{E}} : 2^{\mathcal{V}} \times \mathcal{U}_{\text{ah}} \times 2^{\mathcal{V}} \times \mathcal{R} \to P_{\text{mCRL2}}$ models the provided environment variables D_{disc}, U, J and R. The third function $\mathcal{F}_{\text{time}} : P_{\text{mCRL2}}$ models the value for the variable time. The fourth function $\mathcal{F} : P_{\text{proc}} \times 2^{\mathcal{V}} \times 2^{\mathcal{V}} \to P_{\text{mCRL2}}$ models process term $p \in P_{\text{proc}}$. The implementations are provided later in this section. Process references to the translated process $\mathcal{F}\left(p, D_{\mathsf{disc}}, J\right)$, the global memory management process $X_{\text{mCRL2}}^{Mem([[D_{\mathsf{disc}}]])}$ and the accumulated time process X_{mCRL2}^{Time} are composed in parallel.

The communication function C_M describes the value exchanges between the translated process and the (updated) global variables, and the translated process and the (updated) accumulated time value. The communication function C_C describes the communication between the translated Chi 2.0 channels. The set of allowed actions A_T models Decision 5.5, thereby blocking the modeled deadlocks, non-successful value exchanges for model variables and non-synchronizing channel communications. The set H_M abstracts from the τ_χ labeled actions.

$$\mathcal{F}_{\text{init}}\left(\langle p,\sigma,(D,U,J,R)\rangle\right) = \begin{cases} \mathcal{F}_{\text{Mem}}\left(D_{\mathsf{disc}}\right) \\ \mathcal{F}_{\text{E}}\left(D_{\mathsf{disc}},U,J,R\right) \\ \mathcal{F}_{\text{time}} \\ \textbf{init}\ \tau_{H_M}\left(\nabla_{A_T}\left(\Gamma_{C_M\cup C_C}\left(\begin{array}{l} X_{\text{mCRL2}}^{Mem([[D_{\mathsf{disc}}]])}([[\sigma(D_{\mathsf{disc}})]]) \parallel \\ X_{\text{mCRL2}}^{Time}(time_{\text{H}}(0,0)) \parallel \\ \mathcal{F}\left(p,D_{\mathsf{disc}},J\right) \end{array}\right)\right)\right) \end{cases}$$

where

- $X_{\text{mCRL2}}^{Mem([[D_{\mathsf{disc}}]])}([[\sigma(D_{\mathsf{disc}})]])$ is the process references of the mCRL2 process equation corresponding to the memory process for the variables belonging to D_{disc}, initialized by their corresponding values.
- $X_{\text{mCRL2}}^{Time}(time_{\text{H}}(0,0))$ is the instantiated mCRL2 process by the value $time_{\text{H}}(0,0)$. The process parameter specifies the amount of (hybrid) time that has elapsed.
- $\mathcal{F}\left(p,D_{\mathsf{disc}},J\right)$ provides an mCRL2 specification for process p given D_{disc} and J. The arguments D_{disc} and J restrict the changing variables at the atomic action level that are observed in a *diff*-action.
- $C_M = \{set_{\text{mem}}^{[[v]]} \mid get_{\text{mem}}^{[[v]]} \rightarrow com_{\text{mem}}^{[[v]]}, set_{\text{mem}'}^{[[v]]} \mid get_{\text{mem}'}^{[[v]]} \rightarrow com_{\text{mem}'}^{[[v]]}, set_{\text{time}} \mid get_{\text{time}} \rightarrow com_{\text{time}}, get_{\text{time}'} \mid set_{\text{time}'} \rightarrow com_{\text{time}'} : v \in D_{\mathsf{disc}}\}$ defines (i) the successful value exchange between the translated process and the memory process and (ii) the successful time exchange between the translated process and the time process.
- $C_C = \{send_{[[h]]} \mid recv_{[[h]]} \rightarrow comm_{[[h]]} : h \in \mathcal{H}\}$ defines the communication between the translated Chi 2.0 channels.
- $$\begin{aligned} A_T = &\left\{\Big|_{v\in D_{\mathsf{disc}}}\left(com_{\text{mem}}^{[[v]]} \mid com_{\text{mem}'}^{[[v]]}\right) | com_{\text{time}} \mid com_{\text{time}'} \mid \alpha \mid diff : \alpha \in \mathcal{L}_{\text{basic}} \cup \{\tau_\chi\}\right\} \\ &\cup \left\{\Big|_{v\in D_{\mathsf{disc}}}\left(com_{\text{mem}}^{[[v]]} \mid com_{\text{mem}'}^{[[v]]}\right) | com_{\text{time}} \mid com_{\text{time}'} \mid comm_{[[h]]} \mid diff^2 : h \in \mathcal{H}\right\} \\ &\cup \left\{\Big|_{v\in D_{\mathsf{disc}}}\left(com_{\text{mem}}^{[[v]]} \mid com_{\text{mem}'}^{[[v]]}\right) | com_{\text{time}} \mid com_{\text{time}'} \mid comm_{[[h]]} \mid diff : h \in \mathcal{H}\right\} \\ &\cup \left\{\Big|_{v\in D_{\mathsf{disc}}}\left(com_{\text{mem}}^{[[v]]} \mid com_{\text{mem}'}^{[[v]]}\right) | com_{\text{time}} \mid com_{\text{time}'} \mid \tau_\chi \mid diff^2\right\} \end{aligned}$$
  denotes the allowed actions. Here, $diff^2$ is $diff \mid diff$, and $\Big|_{i\in I} p_i$ is inductively defined by:
  $$\Big|_{i\in\emptyset} p_i = \tau, \qquad \Big|_{i\in I\cup\{k\}} p_i = p_k \mid \Big|_{i\in I\setminus\{k\}} p_i.$$
  The multi-actions that are allowed to happen are defined by A_T. The first subset specifies the set of multi-actions that result from a (hidden) guarded action

update. The second and third subset specify the set of multi-actions that result from a successful communication. The fourth subset specifies the set of multi-actions that result from a hidden successful communication.

- $H_M = \{\tau_\chi\}$ hides internal actions.

The remainder of this chapter uses the expression X^{Chi}_{mCRL2} as the shorthand expression for $\mathcal{F}\left(p, D_{\mathsf{disc}}, J\right)$.

Model Variables

Every syntactic occurrence of a variable scope (including the global variable scope) is modeled as a separate memory process. Due to Decision 5.4, we introduce for each element from the set of model variables V a separate process equation of the form $X^{Mem([\![V]\!])}_{\text{mCRL2}}$. The process stores the values in the process parameters $\vec{x} : \vec{\Lambda}$ in some arbitrary but fixed order such that every $v \in \vec{x}$ is related to $v \in V$. Because all of the local variables are unique, the variables act as an identifier in the process label.

To exchange the values of variables $v \in V$ between the process $X^{Mem([\![V]\!])}_{\text{mCRL2}}$ and the translated Chi 2.0 process X^{Chi}_{mCRL2} we use the actions $set^{[\![v]\!]}_{\text{mem}}, get^{[\![v]\!]}_{\text{mem}'} : \Lambda$. The action $set^{[\![v]\!]}_{\text{mem}}$ provides the value for variable v from the process X^{Chi}_{mCRL2}. The action $get^{[\![v]\!]}_{\text{mem}'}$ retrieves the updated value for variable v from the process X^{Chi}_{mCRL2}. Values are exchanged if the actions synchronize with respectively the actions $get^{[\![v]\!]}_{\text{mem}}$ or $set^{[\![v]\!]}_{\text{mem}'}$, and result in the actions $com^{[\![v]\!]}_{\text{mem}}$ and $com^{[\![v]\!]}_{\text{mem}'}$. The value exchanges are depicted in Figure 5.1.

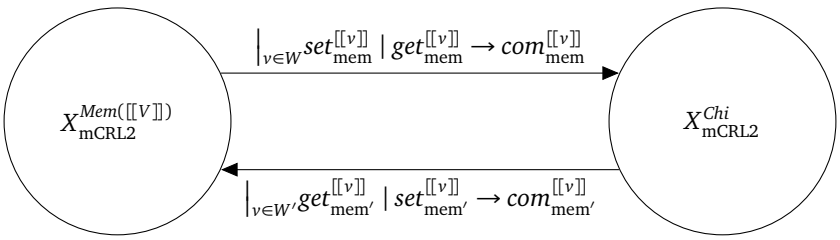


Figure 5.1 Information exchange between a memory process $X^{Mem([\![V]\!])}_{\text{mCRL2}}$ and a translated process X^{Chi}_{mCRL2}

The variables for which the values need to be exchanged are not known in advance. Hence we must allow all subset variable exchanges. So, we specify $W \subseteq V$ that implies that all combinations of subset variables for the com_{mem} actions at the memory process are considered, and $W' \subseteq V$ that implies that all combinations of subset variables for the $com_{\text{mem}'}$ actions at the memory process are considered. The value exchanges for the current values from the memory process to the translated process are restricted

to the required variables. This set is indicated by W. The value exchanges of the updated values from the translated process to the memory process are restricted to those variables for which the values have possibly been changed. This set is indicated by W'. The variables that are not required are not exchanged. Because all values of the model variables need to be visible, we add them separately by $\big|_{v' \in V \setminus W} com_{\text{mem}}^{[\![v']\!]}(\vec{x}_{[\![v']\!]})$ and $\big|_{w' \in V \setminus W'} com_{\text{mem}'}^{[\![w']\!]}(\vec{x}_{[\![w']\!]})$.

With the help of the translation function $\mathcal{F}_{\text{Mem}}$ and the aforementioned considerations, we construct the partial mCRL2 specification:

$$
\begin{aligned}
&\mathcal{F}_{\text{Mem}}(V) = \\
&\quad \textbf{proc}\ X_{\text{mCRL2}}^{Mem([\![V]\!])}(\vec{x} : \vec{\Lambda}) = \\
&\qquad \sum_{W, W' \subseteq V} \sum_{\vec{d} : \vec{\Lambda}} \left(\bigwedge_{w' \in V \setminus W'} \vec{x}_{[\![w']\!]} \approx \vec{d}_{[\![w']\!]} \right) \rightarrow \\
&\qquad \left(\begin{array}{c} \big|_{v \in W} set_{\text{mem}}^{[\![v]\!]}(\vec{x}_{[\![v]\!]}) \mid \big|_{v' \in V \setminus W} com_{\text{mem}}^{[\![v']\!]}(\vec{x}_{[\![v']\!]}) \\ \big|_{w \in W'} get_{\text{mem}'}^{[\![w]\!]}(\vec{d}_{[\![w]\!]}) \mid \big|_{w' \in V \setminus W'} com_{\text{mem}'}^{[\![w']\!]}(\vec{x}_{[\![w']\!]}) \end{array} \right) \cdot X_{\text{mCRL2}}^{Mem([\![V]\!])}(\vec{d});
\end{aligned}
$$

where $\vec{e}_n$ denotes an element of $\vec{e}$ that is associated to variable n. If multiple process access the memory process simultaneously, they mutually have to agree on the selected values. This contract is separately enforced by the communication operator described in Chapter 5.3.7.

Accumulated Time

Decision 5.9 introduces the mCRL2 process X_{mCRL2}^{Time} that stores the accumulated time value. The time value t for the process X_{mCRL2}^{Time} is initially set to $time_{\text{H}}(0, 0)$. If some time passes between two subsequent actions, this is modeled by increasing the value of $\pi_{time}(t)$ and the value of $\pi_{counter}(t)$ is reset to 0. When two actions are subsequently executed without any progression of time, we only increment the value of $\pi_{counter}(t)$.

Time values between the translated process and the time process are exchanged with the help of actions set_{time}, get_{time}, $get_{\text{time}'}$ and $set_{\text{time}'}$. Here, the action set_{time}, from X_{mCRL2}^{Time} and the action get_{time} from the translated process X_{mCRL2}^{Chi} retrieves the current absolute time value and offers it to the translated process term. The action $set_{\text{time}'}$ from X_{mCRL2}^{Chi} and the action $get_{\text{time}'}$ from X_{mCRL2}^{Time} retrieves the updated time value from the translated process term and stores it in process X_{mCRL2}^{Time}. Figure 5.2 depicts exchange of time values.

A retrieve and an update of time are performed by a single multi-actions, i.e., $set_{\text{time}}(t)$ represents the absolute time value at which the last action has been performed and $get_{\text{time}'}(t')$ represents the absolute time value at which the current action

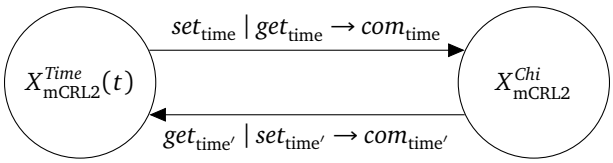


Figure 5.2 Information exchange between a time process X^{Time}_{mCRL2} and a translated process X^{Chi}_{mCRL2}

is performed. Hence we model $\mathcal{F}_{\text{time}}$ by the following mCRL2 specification:

$$
\begin{aligned}
&\mathcal{F}_{\text{time}} = \\
&\quad \textbf{map} \;\; pred_{<} : Time_{\text{H}} \times Time_{\text{H}} \rightarrow \mathbb{B}; \\
&\quad \textbf{var} \;\; t, t' : Time_{\text{H}}; \\
&\quad \textbf{eqn} \;\; pred_{<}(t, t') = (t' \approx time_{\text{H}}(\pi_{time}(t), \pi_{counter}(t) + 1)) \\
&\qquad\qquad \vee (\pi_{time}(t') > \pi_{time}(t) \wedge \pi_{counter}(t') \approx 0); \\
&\quad \textbf{proc} \;\; X^{Time}_{\text{mCRL2}}(t_i{:}Time_{\text{H}}) = \sum_{t':time_{\text{H}}} pred_{<}(t, t') \rightarrow set_{\text{time}}(t) \mid get_{\text{time}'}(t') \cdot X^{Time}_{\text{mCRL2}}(t');
\end{aligned}
$$

Environment Variables

The transformation of the environment variables is performed by evaluating the function $\mathcal{F}_{\text{E}}$, using the environment $(D_{\mathsf{disc}}, U, J, R)$. As D_{disc} and J are required later, the values are passed on to the subsequent translation functions. Because $U \subseteq U_G$ holds, it is modeled when we model U_G. The only environment variable we model is R. Let $R = \{m_1 \mapsto p_1, \ldots, m_n \mapsto p_n\}$ be the process equations that are added to the Chi 2.0 process, then we model for every mapping an mCRL2 process equation:

$$
\begin{aligned}
&\mathcal{F}_{\text{E}}\left(D_{\mathsf{disc}}, U, J, R\right) = \\
&\qquad \textbf{proc} \; [[m_1]] = \mathcal{F}\left(p_1, D_{\mathsf{disc}}, J\right); \\
&\qquad\qquad \vdots \\
&\qquad \textbf{proc} \; [[m_n]] = \mathcal{F}\left(p_n, D_{\mathsf{disc}}, J\right);
\end{aligned}
$$

5.3.6 Atomic Terms

This section describes for each of the atomic Chi 2.0 terms the corresponding mCRL2 notions and their associated ultimate delay functions.

Time Can Progress

The time can progress operator $\mathsf{tcp}\ u$ has a predicate u that allows the passage of time as long as the predicate u stays satisfied, and specifies local urgency. The predicate

becomes *false* at the end point of the delay. To mimic the behavior in the mCRL2 specification we introduce a process that may increase its time value as long as u holds.

The semantics for the operator specifies no action transitions. However, to exchange a time value we require that an action is performed. To model this phenomenon, we construct a multi-action that contains a deadlock action *delta*, thereby taking Decision 5.11 into account. We first exchange the values with the time and the memory process, after which we encapsulate the deadlock action *delta* on the outermost level. Thus we obtain the point in time to which the deadlock occurs, without performing a visible action.

The amount of time that $\mathsf{tcp}\ u$ can delay is determined by the predicate u. The variable $\vec{v} : \vec{\Lambda}$ specifies a chosen vector of values for all discrete model variables. To assert that the vector corresponds to the values of the memory process we synchronize (i.e., retrieve and update) the values for only the relevant values for the model variables w.r.t. $\mathsf{tcp}\ u$. For variables that are irrelevant to the execution of $\mathsf{tcp}\ u$, we select arbitrary values. The variables that are relevant are retrieved by the function $vars : P_{\text{proc}} \to 2^{\mathcal{V}}$, that returns the set of variables that are used in a Chi 2.0 process term. The current time value is exchanged by the value t. The points in time where a deadlock occurs are specified via t'. The maximal amount of time that the process can delay is computed by the ultimate delay function $\Delta^{max}_{time}(\mathsf{tcp}\ u, \pi_{time}(t), \vec{v}) \in \mathbb{R}$. By combining the information, we transform the process term as:

$$\begin{aligned}
&\mathcal{F}\left(\mathsf{tcp}\ u, Vars_D, J\right) = \\
&\quad \sum_{\vec{v}:\vec{\Lambda}} \sum_{t,t':Time_{\mathrm{H}}} \left(\pi_{time}(t') - \pi_{time}(t) \leq \Delta^{max}_{time}\left(\mathsf{tcp}\ u, \pi_{time}(t), \vec{v},\right)\right) \\
&\qquad \to \left(\ \Big|_{z \in vars(\mathsf{tcp}\ u)} \left(get^{[\![z]\!]}_{\text{mem}}(\vec{v}_{[\![z]\!]})\right) \mid get_{\text{time}}(t) \mid delta \mid set_{\text{time}'}(t') \ \right) \cdot t'
\end{aligned}$$

The ultimate delay function Δ^{max}_{time} computes the point in time value $t'' : \mathbb{R}$, starting from the current time $t : \mathbb{R}$ for which all time values up to (including) t'' satisfy predicate u for the ordered set of variables $\vec{v} : \vec{\Lambda}$:

$$\Delta^{max}_{time}(\mathsf{tcp}\ u, t, \vec{v}) = \ \max\{t'' \in \mathbb{R} \mid t \leq t' \wedge t' \leq t'' \wedge \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t')\}$$

Here, the lambda abstraction binds the variables in mCRL2 to the syntactic interpretation of predicate u. Because lambda abstractions are first class citizens in the mCRL2 language, they can be directly used in an mCRL2 specification.

Guarded Action Update

The guarded action update is translated by $\mathcal{F}\left(u \to a : W : r, Vars_D, J\right)$. The guarded action update $u \to a : W : r$ requires (i) the guard u to be satisfied w.r.t. the current values of the model variables, (ii) a new valuation of the model variables should satisfy the update predicate r w.r.t. the old and new valuation, and (iii) the set of model variables for which the values may be changed are provided through $J \cup W$

restricted by $Vars_D$. The guarded action update $u \to a : W : r$ allows arbitrary time transitions for non-urgent actions, i.e., $U(a)$ is *false*. For actions that are urgent, i.e., $U(a)$ is *true*, it only allows the time transition t for which predicate u holds. For all time transitions prior to t predicate u must not hold.

To model (i) and (ii) we first exchange the variables with the memory processes. To retrieve the values for the relevant model variables we use $z \in vars(u \to a : W : r)$. Action $get^{[[z]]}_{\text{mem}}$ and action $set^{[[z]]}_{\text{mem}'}$ respectively receive and send the variables for the model variable z. We retrieve the current time value $t : Time_{\text{H}}$ with the action get_{time} and set the updated time $t' : Time_{\text{H}}$ with action $set_{\text{time}'}$. The set of changing variables (iii) is represented by the mCRL2 action $diff((([[J \cup W]]) \cap Vars_D) \cup \{\text{time}\})$.

To perform the action at time t', both predicate u and predicate r have to hold. To evaluate both we use lambda abstractions. Predicate u reasons on the values of the relevant model variables $\vec{v}$, the time value t at which the last action has been performed, and possible moment in time t' at which the current action is performed. To evaluate predicate u, we construct the lambda abstraction $\lambda_{pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[u]])$ in which the mCRL2 interpretation of u acts as the body. When the action a is performed predicate u has to hold, i.e., modeled by $\lambda_{\vec{v}:\vec{\Lambda},\text{time}':\mathbb{R},\text{time}:\mathbb{R},pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_u(\vec{v},\text{time}'))$. When action a is urgent, we strengthen the condition by demanding that the predicate holds for no time value smaller than the one selected. When the action is non-urgent, then only the predicate u has to hold. This discussion is expressed by*:

$$if(U_G([[a]]), \forall_{t'':\mathbb{R}} \left(\text{time} \leq t'' \wedge t'' < \text{time}' \Rightarrow \neg(pred_u(\vec{v}, t''))\right), true)$$

Predicate r is modeled in a similar way as predicate u, except for the urgency restriction. So, we construct a lambda abstraction, for which the body is the interpretation of r, taking the values of the variables before the action, the variables after the action and the time value at which the predicate r has to hold. We translate the guarded action update as:

$$\begin{aligned}
&\mathcal{F}\left(u \to a : W : r, Vars_D, J\right) = \\
&\sum_{\vec{v}:\vec{\Lambda}} \sum_{\vec{w}:\vec{\Lambda}} \sum_{t,t':Time_{\text{H}}} \\
&\left(\begin{array}{l}
\lambda_{\vec{v}:\vec{\Lambda},\text{time}':\mathbb{R},\text{time}:\mathbb{R},pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_u(\vec{v},\text{time}'))\wedge \\
\quad if(U_G([[a]]), \forall_{t'':\mathbb{R}} \left(\text{time} \leq t'' \wedge t'' < \text{time}' \Rightarrow \neg pred_u(\vec{v}, t'')\right), true) \\
\quad\quad (\vec{v}, \pi_{time}(t'), \pi_{time}(t), \lambda_{pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[u]])) \\
\wedge \\
\lambda_{\vec{v}:\vec{\Lambda},\vec{w}:\vec{\Lambda},\text{time}':\mathbb{R},pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_r(\vec{v},\vec{w},\text{time}')) \\
\quad (\vec{v}, \vec{w}, \pi_{time}(t'), \lambda_{pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[r]]))
\end{array}\right) \\
&\quad \to \left(\begin{array}{l}
\Big|_{z\in vars(u\to a:W:r)} get^{[[z]]}_{\text{mem}}(\vec{v}_{[[z]]}) \mid get_{\text{time}}(t) \mid \\
\qquad [[a]] \mid diff([[((J \cup W) \cap Vars_D) \cup \{\text{time}\}]]) \mid \\
\qquad\quad \Big|_{x\in J\cup W} set^{[[x]]}_{\text{mem}'}(\vec{w}_{[[x]]}) \mid set_{\text{time}'}(t')
\end{array}\right) \circ t'
\end{aligned}$$

Note that the set actions $set^{[[x]]}_{\text{mem}'}$, $x \in J \cup W$ is not restricted to $((J \cup W) \cap Vars_D) \cup \{\text{time}\}$, like the data parameter shown in action *diff*. The restriction is not applied,

*The notation forces the mCRL2 rewriter to evaluate $U_G([[a]])$ prior to the universal quantifier.

because $set^{[[x]]}_{\text{mem}'}$ can exchange values with local memory processes. If we would restrict x to the discrete variables, i.e., $x \in (J \cup W) \cap Vars_D$, it would prevent the value exchange with a local memory processes (Chapter 5.3.7).

The ultimate delay function Δ^{max}_{time} for $u \to a : W : r$ consists of two cases. When an action is urgent, we need to find the first moment in time, that satisfies predicate u. When an action is non-urgent there is no time bound. Hence, the function returns ∞.

$$
\begin{aligned}
&\Delta^{max}_{time}(u \to a : W : r, t, \vec{v}) = \\
&\ \ if(U_G([[a]]), \\
&\qquad \min\{t'' : \mathbb{R} \mid \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t'') \\
&\qquad\qquad \wedge\ \forall_{t':\mathbb{R}} \left(t \leq t' \wedge t' < t'' \Rightarrow \neg\lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t')\right)\}, \infty)
\end{aligned}
$$

Guarded Communication Actions

A guarded send term $u \to h\,!\,e : W : r$ denotes the send of the expression e via channel h. A guarded receive update term $u \to h\,?\,x : W : r$ denotes the receipt of a value via channel h, and stores it into variable x. The process terms are executed when both the guard u and the update predicate r are satisfied.

Because the guarded communication actions share similarities with the guarded action update, their translations are almost identical. Instead of action a, we here require the actions $send_{[[h]]}$ and $recv_{[[h]]}$ to respectively denote the translated actions for the send term ($h!$) and the receive term ($h?$) for the channels $h \in \mathcal{H}$. Because of Decision 5.18 we assume that all communication channels (and also communication channel ends) are non-urgent. Therefore we do not model the urgency restriction. The guarded send communication is translated as:

$$
\begin{aligned}
&\mathcal{F}\left(u \to h\,!\,e : W : r, Vars_D, J\right) = \\
&\quad \sum_{\vec{v}:\vec{\Lambda}} \sum_{\vec{w}:\vec{\Lambda}} \sum_{t,t':Time_{\mathrm{H}}} \\
&\qquad \left(\begin{array}{l}
\quad \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}':\mathbb{R},\mathsf{time}:\mathbb{R},pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_u(\vec{v}, \mathsf{time}')) \\
\qquad\qquad (\vec{v}, \pi_{time}(t'), \pi_{time}(t), \lambda_{pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[u]])) \\
\wedge \\
\quad \lambda_{\vec{v}:\vec{\Lambda},\vec{w}:\vec{\Lambda},\mathsf{time}':\mathbb{R},pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_r(\vec{v}, \vec{w}, \mathsf{time}')) \\
\qquad\quad (\vec{v}, \vec{w}, \pi_{time}(t'), \lambda_{pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[r]]))
\end{array}\right) \\
&\qquad \to \left(\begin{array}{l}
\ \Big|_{z\in vars(u \to h\,!\,e:W:r)} get^{[[z]]}_{\text{mem}}(\vec{v}_{[[z]]}) \mid get_{\text{time}}(t) \\
\mid send_{[[h]]}((\lambda_{\vec{x}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}[[e]])(\vec{v}, \pi_{time}(t'))) \\
\mid diff([[((W \cup J) \cap Vars_D) \cup \{\mathsf{time}\}]]) \\
\mid \Big|_{x\in J\cup W} set^{[[x]]}_{\text{mem}'}(\vec{w}_{[[x]]}) \mid set_{\text{time}'}(t')
\end{array}\right) \circ t'
\end{aligned}
$$

The guarded receive communication is translated as:

$$
\begin{array}{l}
\mathcal{F}\left(u \rightarrow h\,?\,x : W : r, Vars_D, J\right) = \\
\sum\limits_{\vec{v}:\vec{\Lambda}} \sum\limits_{\vec{w}:\vec{\Lambda}} \sum\limits_{t,t':Time_{\mathrm{H}}} \\
\left(\begin{array}{l}
\quad \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}':\mathbb{R},\mathsf{time}:\mathbb{R},pred_u:\vec{\Lambda}\times\mathbb{R}\rightarrow\mathbb{B}}(pred_u(\vec{v},\mathsf{time}')) \\
\qquad (\vec{v}, \pi_{time}(t'), \pi_{time}(t), \lambda_{pred_r:\vec{\Lambda}\times\mathbb{R}\rightarrow\mathbb{B}}([[u]])) \\
\wedge \\
\quad \lambda_{\vec{v}:\vec{\Lambda},\vec{w}:\vec{\Lambda},\mathsf{time}':\mathbb{R},pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\rightarrow\mathbb{B}}(pred_r(\vec{v},\vec{w},\mathsf{time}')) \\
\qquad (\vec{v}, \vec{w}, \pi_{time}(t'), \lambda_{pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\rightarrow\mathbb{B}}([[r]]))
\end{array}\right) \\
\rightarrow \left(\begin{array}{l}
\Big|_{z\in vars(u \rightarrow h\,?\,x:W:r)} get^{[[z]]}_{\mathrm{mem}}(\vec{v}_{[[z]]}) \mid get_{\mathrm{time}}(t) \mid recv_{[[h]]}(\vec{w}_{[[x]]}) \\
\mid diff([[((W \cup J \cup \{x\}) \cap Vars_D) \cup \{\mathsf{time}\}]]) \\
\mid \Big|_{y\in J\cup W\cup\{x\}} set^{[[y]]}_{\mathrm{mem}'}(\vec{w}_{[[y]]}) \mid set_{\mathrm{time}'}(t')
\end{array}\right) \triangleleft t'
\end{array}
$$

Decision 5.18 assumes that all communication channels are non-urgent. So, the ultimate delay functions for any of the communicating channels ends will be ∞. Hence:

$$
\begin{array}{l}
\Delta^{max}_{time}(u \rightarrow h\,!\,e : W : r, t, \vec{v}) = \infty \\
\Delta^{max}_{time}(u \rightarrow h\,?\,x : W : r, t, \vec{v}) = \infty
\end{array}
$$

Recursion Variable Process Term

The recursion variable process term models repetition. If $X \in \mathcal{M}$ denotes a recursion variable, then variable X can do whatever the process term of its definition can do. The process term is either defined through the environment variable R or by one of the recursion scope operators $|[_{\mathrm{R}}\vec{X} \mapsto \vec{q} :: p\ ]|$, and X corresponds to a process label in the translated process. So, we can directly use $[[X]]$ as a process reference.

$$
\mathcal{F}\left(X, Vars_D, J\right) = [[X]]
$$

The ultimate delay function for the recursion variable process, is not affected by translating the recursion variable process term. Therefore, if $X \mapsto p$, then we model:

$$
\Delta^{max}_{time}(X, t, \vec{v}) = \Delta^{max}_{time}(p, t, \vec{v})
$$

5.3.7 Process Terms

The translation for the Chi 2.0 process terms are explained next. This also incorporates the specifications for the corresponding ultimate delay functions.

Sequential Composition Operator

The sequential composition of the process terms p and q, written as $p;q$, behaves as process term p until p terminates, and subsequently behaves as process term q. To

express this behavior in the mCRL2 specification we use the sequential composition operator '·':

$$\mathcal{F}\left(p;q, Vars_D, J\right) = \\ \mathcal{F}\left(p, Vars_D, J\right) \cdot \mathcal{F}\left(q, Vars_D, J\right)$$

The ultimate delay function for $p;q$ is calculated from the actions that can be performed by p. Hence, we specify:

$$\Delta^{max}_{time}(p;q, t, \vec{v}) = \Delta^{max}_{time}(p, t, \vec{v})$$

Alternative Composition Operator

The alternative composition operator applied to process terms p and q, denoted $p \,[\!]\, q$, describes the non-deterministic choice between the behaviors of p and q. The non-deterministic choice is in the mCRL2 language denoted by '+'. Since the languages have different non deterministic notions (i.e., strong time deterministic versus weak time deterministic), a straightforward transformation of the alternative composition operator is not possible. Hence, we take Decision 5.12 into account.

With the help of the function Δ^{max}_{time} we determine the delay that the alternative composition may perform. The maximal delay is computed from the current values of the model variables, the value of the current time and the values of the updated model variables. Hence, the we decorate the alternatives with the synchronizing actions that exchange the current and the updated values for both the time and the memory processes.

To assert that the decorated actions correspond to the actions performed by the translated process terms, we add the communication operator. This function takes two exchange actions that have equal values (e.g., the $get^{[[\cdot]]}_{\text{mem}}$ actions) and produces a single exchange action. Multiple value receives and send requests for the same variable are prevented by A_T (Chapter 5.3.5). Hence we introduce a communication operator. By applying the communication they mutually agree on a value. When a communication cannot be applied, either only one value exchange for a variable is required, or different values have been selected and will therefore be blocked by A_T.

To exchange the current value of the model variables for the processes p and q, we specify $\big|_{x \in vars(p) \cup vars(q)} get^{[[x]]}_{\text{mem}}(\vec{v}_{[[x]]})$. A choice dictates different futures. Therefore it is possible that a variable can have different values for a different futures. Hence we need to updated variables for each of the branches separately. When process term p is executed, we update the values of the model variables by $\big|_{y \in vars^+(p)} set^{[[y]]}_{\text{mem}'}(\vec{w}_{[[y]]})$. When process term q is executed, we update the values of the model variables by $\big|_{y \in vars^+(q)} set^{[[y]]}_{\text{mem}'}(\vec{w}_{[[y]]})$. Note that $vars^+ : P_{\text{proc}} \rightarrow 2^{\mathcal{V}}$ returns the set of model variables that are updated by executing the process term.

With the aforementioned constructs we translate $p \,[\!]\, q$ as:

$$\begin{aligned}
&\mathcal{F}\left(p \,[\!]\, q, Vars_D, J\right) = \\
&\quad \sum_{\vec{v}:\vec{\Lambda}} \sum_{\vec{w}:\vec{\Lambda}} \sum_{t,t':Time_{\mathrm{H}}} \left(\pi_{time}(t') - \pi_{time}(t) \leq \Delta^{max}_{time}(p \,[\!]\, q, \pi_{time}(t), \vec{v})\right) \rightarrow \\
&\qquad \Gamma_{\{get^{[\![v]\!]}_{\mathrm{mem}} | get^{[\![v]\!]}_{\mathrm{mem}} \rightarrow get^{[\![v]\!]}_{\mathrm{mem}}, set^{[\![v]\!]}_{\mathrm{mem'}} | set^{[\![v]\!]}_{\mathrm{mem'}} \rightarrow set^{[\![v]\!]}_{\mathrm{mem'}}, get_{\mathrm{time}} | get_{\mathrm{time}} \rightarrow get_{\mathrm{time}}, set_{\mathrm{time}} | set_{\mathrm{time}} \rightarrow set_{\mathrm{time}} : v \in \mathcal{V}\}} \\
&\qquad\quad \left(\begin{array}{l} \Big|_{x \in vars(p) \cup vars(q)} get^{[\![x]\!]}_{\mathrm{mem}}(\vec{v}_{[\![x]\!]}) \mid get_{\mathrm{time}}(t) \mid set_{\mathrm{time}}(t') \mid \\ \left(\begin{array}{l} \quad \mathcal{F}\left(p, Vars_D, J\right) \mid \Big|_{y \in vars^+(p)} set^{[\![y]\!]}_{\mathrm{mem'}}(\vec{w}_{[\![y]\!]}) \\ + \\ \quad \mathcal{F}\left(q, Vars_D, J\right) \mid \Big|_{y \in vars^+(q)} set^{[\![y]\!]}_{\mathrm{mem'}}(\vec{w}_{[\![y]\!]}) \end{array} \right) \end{array} \right) \circ t'
\end{aligned}$$

The ultimate delay function for the alternative composition operator is defined as:

$$\begin{aligned}
&\Delta^{max}_{time}(p \,[\!]\, q, t, \vec{v}) = \\
&\qquad \min(\Delta^{max}_{time}(p, t, \vec{v}), \Delta^{max}_{time}(q, t, \vec{v}))
\end{aligned}$$

Parallel Composition Operator

The parallel composition for process terms p and q, denoted by $p \parallel q$ describes the interleaving behavior of the process terms p and q. The mCRL2 language uses the same operator to denote parallelism. The semantics for the parallel operator of the mCRL2 language differs, as it *interleaves* and *synchronizes* the actions that are performed by p and q. So, when we model $p \parallel q$ in the mCRL2 language, it gives rise to additional multi-actions. Hence, the behavior is restricted by applying Decision 5.7.

To model Decision 5.7 we subsequently apply the communication operator and the allow operator. The communication denotes the synchronizing actions for the channel communication is modeled by C. The mutual value exchange for both model variables and time is modeled by C'. The allow operator only allows actions that comply to the signature of a modeled Chi 2.0 action transition. The allowed actions are modeled via the set A, that consists of a set of $get^{[\![v]\!]}_{\mathrm{mem}}$ actions (where $v \in V$ for which $V \subseteq \mathcal{V}$ holds), one action that corresponds to a Chi 2.0 action, one or two *diff* actions (depending on the number synchronizing actions, which is at most two), and a set of $set^{[\![v']\!]}_{\mathrm{mem'}}$ actions (where $v' \in V'$ for which $V' \subseteq \mathcal{V}$ holds).

So, we specify the translation of $p \parallel q$ as:

$$\begin{aligned}
&\mathcal{F}\left(p \parallel q, Vars_D, J\right) = \\
&\quad \nabla_A \left(\Gamma_{C \cup C'} \left(\mathcal{F}\left(p, Vars_D, J\right) \parallel \mathcal{F}\left(q, Vars_D, J\right) \right)\right)
\end{aligned}$$

where

- the set of allowed actions is described by:

$$A = \left\{ \begin{array}{l} \bigcup_{W,W' \subseteq \mathcal{V}} \left(\Big|_{v \in W} get^{[\![v]\!]}_{\mathrm{mem}} \mid \alpha \mid diff \mid \Big|_{w \in W'} set^{[\![w]\!]}_{\mathrm{mem'}} | get_{\mathrm{time}} | set_{\mathrm{time'}} \right), \\ \bigcup_{W,W' \subseteq \mathcal{V}} \left(\Big|_{v \in W} get^{[\![v]\!]}_{\mathrm{mem}} \mid \eta \mid diff \mid diff \mid \Big|_{w \in W'} set^{[\![w]\!]}_{\mathrm{mem'}} | get_{\mathrm{time}} | set_{\mathrm{time'}} \right) \end{array} \right\}$$

where $\alpha \in \mathcal{L}_{\text{basic}} \cup \{\tau_\chi\} \cup \{send_{[\![h]\!]}, recv_{[\![h]\!]} \ : \ h \in \mathcal{H}\}$, $\eta \in \{comm_{[\![h]\!]} \ : \ h \in \mathcal{H}\}$

- the set of synchronizing communication actions of a translated Chi 2.0 process is described by:

$$C = \{send_{[\![h]\!]} \mid recv_{[\![h]\!]} \rightarrow comm_{[\![h]\!]} \ : \ h \in \mathcal{H}\}$$

- the set of synchronizing communication actions that eliminates duplicate value exchanges during a communication is described by:

$$C' = \left\{ \begin{array}{c} get_{\text{mem}}^{[\![v]\!]} \mid get_{\text{mem}}^{[\![v]\!]} \rightarrow get_{\text{mem}}^{[\![v]\!]}, set_{\text{mem}'}^{[\![v]\!]} \mid set_{\text{mem}'}^{[\![v]\!]} \rightarrow set_{\text{mem}'}^{[\![v]\!]}, \\ get_{\text{time}} \mid get_{\text{time}} \rightarrow get_{\text{time}}, set_{\text{time}'} \mid set_{\text{time}'} \rightarrow set_{\text{time}'} \end{array} \ : v \in \mathcal{V} \right\}$$

The ultimate delay function for the parallel composition operator is defined as:

$$\begin{array}{l} \Delta_{time}^{max}(p \parallel q, t, \vec{v}) = \\ \qquad \min(\Delta_{time}^{max}(p, t, \vec{v}), \Delta_{time}^{max}(q, t, \vec{v})) \end{array}$$

Channel encapsulation operator

The behavior of a channel encapsulation operator $\partial_{H'}$ applied to a process term p, informally states that send and receive actions from the set H' cannot propagate beyond the scope of the operator. Therefore, these communication ends are blocked. The behavior is modeled using the mCRL2 encapsulation operator. So, we model $\partial_B(p)$, where $B = \{send_{[\![h]\!]}, recv_{[\![h]\!]} : h \in H'\}$ specifies the set of communication ends that need to be blocked. Thus we model $\partial_{H'}(p)$ as:

$$\begin{array}{c} \mathcal{F}\left(\partial_{H'}(p), Vars_D, J\right) = \\ \partial_B\left(\mathcal{F}\left(p, Vars_D, J\right)\right) \end{array}$$

The ultimate delay function for the channel encapsulation operator is defined as:

$$\Delta_{time}^{max}(\partial_{H'}(p), t, \vec{v}) = \Delta_{time}^{max}(p, t, \vec{v})$$

Variable Scope Operator

By means of the variable scope operator $|[_{\mathrm{V}}\, d_{\vec{x}}, \sigma_{\vec{x}} :: p\ ]|$ local model variables are introduced. Here, $d_{\vec{x}}$ denotes a dynamic type mapping with domain $dom(d_{\vec{x}}) = \{x_1, \ldots, x_n\}$ that corresponds to $\vec{x}$, and $\sigma_{\vec{x}}$ denotes a local valuation for the state variables of the domain of $d_{\vec{x}}$ ($dom(\sigma_{\vec{x}}) = dom(d_{\vec{x}})$).

Because of Decision 5.13, all model variables that are defined by $dom(d_{\vec{x}})$ are unique with respect to all other (local) model variables, which implies that no renaming is required. To model the variable scope operator we introduce an mCRL2 process equation that models local the memory management, which is provided through $\mathcal{F}_{\text{Mem}}\left(dom(d_{\vec{x}})\right)$. To ensure that the values are exchanged we add a reference to the

process definition, i.e., $X_{\text{mCRL2}}^{Mem([\![dom(d_{\vec{x}})]\!])}$. To enforce the variable binding to the most local variables, we enforce that process p exchanges values with the introduced memory process $X_{\text{mCRL2}}^{Mem([\![dom(d_{\vec{x}})]\!])}$ only for those variables that are locally introduced. The value exchanges between the memory process $X_{\text{mCRL2}}^{Mem([\![dom(d_{\vec{x}}]\!]))}$ and the translated process X_{mCRL2}^{Chi} are performed in similar ways as we have seen in Chapter 5.3.5. The semantics of the Chi 2.0 language states that local value exchanges are non-observable. Hence, the resulting successful value exchanges $com_{\text{mem}}^{[\![v]\!]}, com_{\text{mem}'}^{[\![v]\!]}$ are hidden. The actions $get_{\text{mem}}^{[\![v]\!]}, set_{\text{mem}'}^{[\![v]\!]}, set_{\text{mem}}^{[\![v]\!]}$ or $get_{\text{mem}'}^{[\![v]\!]}$ are blocked, to prohibit the exchange of values with other surrounding memory processes (e.g., when a variable scope is nested inside a recursion scope). The translation for the variable scope operator is defined as:

$$\mathcal{F}\left(|[_{\text{V}}\, d_{\vec{x}}, \sigma_{\vec{x}} :: p\ ]|, Vars_D, J\right) = \\ \tau_{H'_M}\left(\partial_{B'_M}\left(\Gamma_{C'_M}\left(X_{\text{mCRL2}}^{Mem([\![dom(d_{\vec{x}})]\!])}\left([\![\sigma_{\vec{x}}(dom(d_{\vec{x}}))]\!]\right) \parallel \mathcal{F}\left(p, Vars_D, J\right)\right)\right)\right)$$

where

- the communication of the successful value exchanges is specified by:

$$C'_M = \{get_{\text{mem}}^{[\![v]\!]} \mid set_{\text{mem}}^{[\![v]\!]} \rightarrow com_{\text{mem}}^{[\![v]\!]}, set_{\text{mem}'}^{[\![v]\!]} \mid get_{\text{mem}'}^{[\![v]\!]} \rightarrow com_{\text{mem}'}^{[\![v]\!]} : v \in dom(d_{\vec{x}})\}$$

- the non-successful communications are blocked by:

$$B'_M = \{get_{\text{mem}}^{[\![v]\!]}, set_{\text{mem}}^{[\![v]\!]}, get_{\text{mem}'}^{[\![v]\!]} set_{\text{mem}'}^{[\![v]\!]} : v \in dom(d_{\vec{x}})\}$$

- the abstraction on the local value exchanges is defined as:

$$H'_M = \{com_{\text{mem}}^{[\![v]\!]}, com_{\text{mem}'}^{[\![v]\!]} : v \in dom(d_{\vec{x}})\}$$

The additional process equation that results from $\mathcal{F}_{\text{Mem}}\left(dom(d_{\vec{x}})\right)$ is added separately to the mCRL2 specification.

The variable scope introduces locally initialized variables. When computing the ultimate delay function, these initialized variables need to be added. Hence, we update the values for the corresponding variables in $\vec{v}$. The update of variable i in $\vec{v}$ by value w is represented by $\vec{v}[i \mapsto w]$. A collection of updates is represented by a subscript after the last square bracket. Hence, we model the ultimate delay function for the variable scope as:

$$\Delta_{time}^{max}(|[_{\text{V}}\, d_{\vec{x}}, \sigma_{\vec{x}} :: p\ ]|, t, \vec{v}) = \Delta_{time}^{max}(p, t, \vec{v}[i \mapsto \sigma_{\vec{x}}(i)]_{i \in dom(d_{\vec{x}})})$$

Recursion Scope Operator

The recursion scope operator $|[_{\text{R}}\, \{\vec{X} \mapsto \vec{q}\} :: p\ ]|$ allows local recursion. A recursion scope operators contains a mapping $\vec{X} \mapsto \vec{q}$, that expresses that every recursion variable X_i maps to a process term q_i, $1 \leq i \leq N$.

Under the assumption of Decision 5.14, stating that every mode is unique, we translate the recursion scope operator. So, for every mapping we introduce a (global) mCRL2 process equation, where every Chi 2.0 mode corresponds to a process label, and an associated process term corresponds to the translation of that term. So, we model $|[_{\mathrm{R}}\ \{\vec{X} \mapsto \vec{q}\} :: p\ ]|$ as:

$$= \begin{array}{l} \mathcal{F}\left(|[_{\mathrm{R}}\ \{\vec{X} \mapsto \vec{q}\} :: p\ ]|, Vars_D, J\right) \\ \mathcal{F}\left(p, Vars_D, J\right) \end{array}$$

Additionally we model:

$$\begin{array}{c} \textbf{proc}\ [[\vec{X}_1]] = \mathcal{F}\left(\vec{q}_1, Vars_D, J\right); \\ \vdots \\ \textbf{proc}\ [[\vec{X}_n]] = \mathcal{F}\left(\vec{q}_n, Vars_D, J\right); \end{array}$$

The ultimate delay function for the recursion scope operator is specified as:

$$\Delta^{max}_{time}(|[_{\mathrm{R}}\ \{\vec{X} \mapsto \vec{q}\} :: p\ ]|, t, \vec{v}) = \Delta^{max}_{time}(p, t, \vec{v})$$

Action Scope Operator

The action scope operator $|[_{\mathrm{A}}\ U_A :: p\ ]|$ allows local basic actions, which are hidden for the surrounding processes. Because of Decision 5.15, we know that all actions are unique. As the global urgency mapping U_G has been defined in Chapter 5.3.4 and $U_A \subseteq U_G$, the urgency mapping U_A is already present in the resulting translation. For reasons given in Decision 5.8, the internal actions are renamed to τ_χ. Based on these decisions we model the action scope operator as:

$$\begin{array}{c} \mathcal{F}\left(|[_{\mathrm{A}}\ U_A :: p\ ]|, Vars_D, J\right) = \\ \rho_{Ren}\left(\mathcal{F}\left(p, Vars_D, J\right)\right) \end{array}$$

where the rename function is defined as: $Ren = \{[[a]] \rightarrow \tau_\chi : a \in dom(U_A)\}$. The corresponding ultimate delay function is defined as:

$$\Delta^{max}_{time}(|[_{\mathrm{A}}\ U_A :: p\ ]|, t, \vec{v}) = \Delta^{max}_{time}(p, t, \vec{v})$$

Channel Scope Operator

The channel scope operator $|[_{\mathrm{A}}\ H_L :: p\ ]|$ defines local channels. Successful communications survive outside the scope as internal actions. Non-successful communications are blocked.

In Chapter 5.3.4 we have seen that $H_L \subseteq U_G$. Hence, all channels are already defined globally. Based on Decision 5.16, we know that all channels actions are globally unique, so there is no need to replace any of the channel labels. Successful communications are renamed to τ_χ, for similar reasons that we have seen in e.g., the hiding

of actions in the action scope operator. Non-successful communications are blocked with the help of $send_{[\![h]\!]}, recv_{[\![h]\!]}$ $(h \in dom(H_L))$. The translation for $|[_{\mathrm{A}} H_L :: p\ ]|$ is then defined as:

$$\mathcal{F}\left(|[_{\mathrm{A}} H_L :: p\ ]|, Vars_D, J\right) = \tau_{H_{Com}}\left(\partial_B\left(\Gamma_{Com}\left(\mathcal{F}\left(p, Vars_D, J\right)\right)\right)\right)$$

where

- the communication function is defined as:

$$Com = \{send_{[\![h]\!]} \mid recv_{[\![h]\!]} \rightarrow comm_{[\![h]\!]} : h \in dom(H_L)\}$$

- the blocking actions are defined as:

$$B = \{send_{[\![h]\!]}, recv_{[\![h]\!]} : h \in dom(H_L)\}$$

- the communicating actions are hidden according to:

$$H_{Com} = \{comm_{[\![h]\!]} : h \in dom(H_L)\}$$

The ultimate delay function for the channel scope operator is defined as:

$$\Delta^{max}_{time}(|[_{\mathrm{H}} H' :: p\ ]|, t, \vec{v}) = \Delta^{max}_{time}(p, t, \vec{v})$$

5.4 Additional Considerations

Although we assume that the translation preserves the intended semantics and the result is a valid mCRL2 specification, we here provide considerations that translate a larger subset if we slightly tweak the Chi 2.0 input. We also provide considerations, because some translated Chi 2.0 notions prevent exhaustive simulations. The considerations are provided for the subset described in Chapter 5.2.

5.4.1 Valuation with Undefined Variables

The original Chi 2.0 language extends the valuation with undefined variables. $\Sigma_\perp = \mathcal{V} \mapsto \Lambda_\perp$ denotes the set of all variable valuations with undefined variables. Variables may have the undefined 'value' $\perp$ $(\perp \notin \Lambda)$. A valuation that contains undefined variables is defined by $\Lambda_\perp = \Lambda \cup \{\perp\}$. For presentation purposes we assume all variables are defined. It should be obvious that a valuation with undefined variables poses no problem for the translation.

5.4.2 Set of Changing Variables

The changing sets of variables (represented by W) are provided in atomic process terms. We advice to restrict the use of W to the smallest set of variables contained in predicate u and update function r. If W represents a larger set of variables, arbitrary values can be assigned to those. If the sort of the variable is represented by an infinite domain (e.g., $\mathbb{N}$), all possible values are considered, which renders any exhaustive simulation useless.

5.4.3 Time

The ultimate delay function assumes that we can compute the value for an infinite delay ∞. Models that are not restricted by an upper time bound and define non-urgent actions are less suitable for an analysis. Therefore, we advise to translate (and simulate) models for which the delays are restricted by some upper-bound time value.

5.4.4 Urgency on Channel Ends

Decision 5.18 assumes that channels are non-urgent in a Chi 2.0 specification. We here present an alternative solution that allows for urgency on communicating channels. The urgency is defined for channel ends in contrast to successfully communicating channels. The solution is reasonable, since many architectures define only one end to be urgent.

Incorporating the change requires a slight change to the syntax and the semantics of the Chi 2.0 language. Originally, $\mathcal{H}$ denotes the set of channel names for which urgency is defined for successful communicating channels with help of U_h. The suggested solution defines the successful communication channels for the sending and receiving channel ends, respectively $\mathcal{H}_!$ and $\mathcal{H}_?$. Their urgency are specified by U_h. The suggested syntactical change is illustrate in Table 5.1. The left column illustrates the proposed typing for the urgent channels. The right column illustrates the syntax that could be accommodated.

Current notation	**Proposed notation**
$U_h : \mathcal{H} \rightarrow \mathbb{B}$	$U_h : (\mathcal{H}_! \cup \mathcal{H}_?) \rightarrow \mathbb{B}$
$\{h \rightarrow \mathit{true}\}$	$\{h! \rightarrow \mathit{true}, h? \rightarrow \mathit{false}\}$ or $\{h! \rightarrow \mathit{false}, h? \rightarrow \mathit{true}\}$

Table 5.1 Suggested definition for urgency on channel ends

The translation from Chapter 5.3.6 assumes that all channels are non urgent. If we incorporate the individual urgency on channel ends, we have to alter the transformation rule and the ultimate delay function. The sending and receiving transformation rules have to respectively by edited, by adding the black colored line and substitute $\mathbb{H}$

by either $h\,!$ or $h\,?$. The gray lines state the parts for the transformations that already have been provided.

$$\left(\begin{array}{l}
\quad \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}':\mathbb{R},\mathsf{time}:\mathbb{R},pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_u(\vec{v},\mathsf{time}')) \\
\qquad \wedge\ if(U_G([[\mathbb{H}]]), \forall_{t'':\mathbb{R}} \left(\mathsf{time} \leq t'' \wedge t'' < \mathsf{time}' \Rightarrow \neg(pred_u(\vec{v},t''))\right), true) \\
\qquad\quad (\vec{v}, \pi_{time}(t'), \pi_{time}(t), \lambda_{pred_u:\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[u]])) \\
\wedge \\
\quad \lambda_{\vec{v}:\vec{\Lambda},\vec{w}:\vec{\Lambda},\mathsf{time}':\mathbb{R},pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}(pred_r(\vec{v},\vec{w},\mathsf{time}')) \\
\qquad (\vec{v},\vec{w},\pi_{time}(t'), \lambda_{pred_r:\vec{\Lambda}\times\vec{\Lambda}\times\mathbb{R}\to\mathbb{B}}([[r]]))
\end{array}\right)$$

The ultimate delay functions for the communication process have to be updated as well. Hence we redefine the ultimate delay functions accordingly:

$$\begin{array}{l}
\Delta^{max}_{time}(u \to h\,!\,e : W : r, t, \vec{v}) = \\
\quad if(U_G([[h\,!]]), \\
\qquad \min\{t'' : \mathbb{R} \mid \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t'') \\
\qquad\qquad\qquad \wedge\ \forall_{t':\mathbb{R}} \left(t \leq t' \wedge t' < t'' \Rightarrow \neg\lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t')\right)\}, \infty)
\end{array}$$

$$\begin{array}{l}
\Delta^{max}_{time}(u \to h\,?\,x : W : r, t, \vec{v}) = \\
\quad if(U_G([[h\,?]]), \\
\qquad \min\{t'' : \mathbb{R} \mid \lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t'') \\
\qquad\qquad\qquad \wedge\ \forall_{t':\mathbb{R}} \left(t \leq t' \wedge t' < t'' \Rightarrow \neg\lambda_{\vec{v}:\vec{\Lambda},\mathsf{time}:\mathbb{R}}([[u]])(\vec{v}, t')\right)\}, \infty)
\end{array}$$

5.5 Examples

This section presents four models with their corresponding transformations that validate the translation. Although the resulting models are *valid*, some of the models cannot be directly executed due to tool restrictions. To circumvent these restrictions, post processing steps are required. The required processing steps are stated in a separate discussion after the transformation. The actual models are provided in Appendix B.2.

The presented models share common aspects, e.g., action declarations, variable labels and the time sort. The common concepts are provided first and hold for all examples, unless stated otherwise.

The common sort declarations are provided first. The structured sort $\mathcal{L}$ specifies the actions that are provided by the Chi 2.0 specifications. The sort $\mathcal{V}$ specifies the modeled Chi 2.0 variables. We assume that Λ is restricted to the sort $\mathbb{B}$. The sort $Time_{\mathrm{H}}$ denotes the sort for the time domain with micro steps. We introduce an alias sort $\mathcal{S}_{Time}$ to conveniently adapt the resolution of the time domain.

$$\begin{array}{ll}
\textbf{sort} & \mathcal{L} = \textbf{struct}\ a \mid b \mid send_c \mid send_c; \\
& \mathcal{V} = \textbf{struct}\ s \mid \mathsf{time}; \\
& \mathcal{S}_{Time} = \mathbb{R}; \\
& Time_{\mathrm{H}} = \textbf{struct}\ time_{\mathrm{H}}(\pi_{time} : \mathcal{S}_{Time}, \pi_{counter} : \mathbb{N});
\end{array}$$

The action declaration declares the actions a, b, and $send_c, recv_c, comm_c : \mathbb{B}$. The Chi 2.0 specification only models a variable $s : \mathbb{B}$. We only declare set^s_{mem}, get^s_{mem}, com^s_{mem}, $set^s_{\text{mem}'}$, $get^s_{\text{mem}'}$ and $com^s_{\text{mem}'}$. Furthermore, we add the required auxiliary actions τ_χ and *diff*.

act $a, b, \tau_\chi;$
$send_c, recv_c, comm_c : \mathbb{B};$
$set_{\text{time}}, get_{\text{time}}, set_{\text{time}'}, get_{\text{time}'} : Time_{\text{H}};$
$set^s_{\text{mem}}, get^s_{\text{mem}}, com^s_{\text{mem}}, set^s_{\text{mem}'}, get^s_{\text{mem}'}, com^s_{\text{mem}'} : \mathbb{B};$
$diff : Set(\mathcal{V});$

All of the examples use the same urgency mapping. Therefore U_G is defined commonly. The urgency for channels is defined for channel ends (Chapter 5.4.4). We assume that a and $send_c$ are the only urgent actions.

map $U_G : \mathcal{L} \to \mathbb{B};$
eqn $U_G(a) = true;$
$U_G(b) = false;$
$U_G(send_c) = true;$
$U_G(recv_c) = false;$

The initialization is derived from the transformation scheme. Note that $\{s \mapsto true\}$ is the valuation that initializes the global memory process $X^{Mem(\{s\})}_{\text{mCRL2}}$ for all of the examples whenever required.

proc
$$\begin{aligned} X^{Mem(\{s\})}_{\text{mCRL2}}(s : \mathbb{B}) = & \\ & \sum_{s':\mathbb{B}} set^s_{\text{mem}}(s) \mid get^s_{\text{mem}'}(s') \cdot X^{Mem(\{s\})}_{\text{mCRL2}}(s') \\ & + \quad set^s_{\text{mem}}(s) \mid com^s_{\text{mem}'}(s) \cdot X^{Mem(\{s\})}_{\text{mCRL2}}(s) \\ & + \sum_{s':\mathbb{B}} com^s_{\text{mem}}(s) \mid com^s_{\text{mem}'}(s') \cdot X^{Mem(\{s\})}_{\text{mCRL2}}(s') \\ & + \quad com^s_{\text{mem}}(s) \mid com^s_{\text{mem}'}(s) \cdot X^{Mem(\{s\})}_{\text{mCRL2}}(s); \end{aligned}$$

init
$$\begin{aligned} & \tau_{\{\tau_\chi\}} \\ & \nabla \left\{ \begin{array}{l} com_{\text{time}} | com^s_{\text{mem}} | a | diff | com^s_{\text{mem}'} | com_{\text{time}'},\ com_{\text{time}} | com^s_{\text{mem}} | b | diff | com^s_{\text{mem}'} | com_{\text{time}'}, \\ com_{\text{time}} | com^s_{\text{mem}} | \tau_c | diff | com^s_{\text{mem}'} | com_{\text{time}'},\ com_{\text{time}} | com^s_{\text{mem}} | comm_c | diff | diff | com^s_{\text{mem}'} | com_{\text{time}'} \end{array} \right\} \\ (\quad & \Gamma_{\{set_{\text{time}} | get_{\text{time}} \to com_{\text{time}}, set_{\text{time}'} | get_{\text{time}'} \to com_{\text{time}'}, set^s_{\text{mem}} | get^s_{\text{mem}} \to com^s_{\text{mem}}, set^s_{\text{mem}'} | get^s_{\text{mem}'} \to com^s_{\text{mem}'}\}} \\ & X^{Mem(\{s\})}_{\text{mCRL2}}(true) \parallel X^{Time}_{\text{mCRL2}}(time_{\text{H}}(0,0)) \parallel X^{Chi}_{\text{mCRL2}}); \end{aligned}$$

5.5.1 Guarded Action Update Example

The first example is taken from [BHR+08]:

$$\langle \mathsf{time} \geq 1 \to a : \emptyset : true, \{\mathsf{time} \mapsto 0\}, (\{\mathsf{time} \mapsto \mathsf{cont}\}, \{a \mapsto true\}, \emptyset, \emptyset) \rangle$$

The process delays until time point 1 is reached. At that point in time, the guard ($\mathsf{time} \geq 1$) becomes *true*. Hereto, the action a becomes enabled. The action label is

declared urgent via the urgency mapping $\{a \to \mathit{true}\}$. This means that it is impossible to delay the action at time point 1. Hence, the guarded action update statement "$\mathsf{time} \geq 1 \to a : \emptyset : \mathit{true}$" must be executed, and terminates subsequently.

If we apply the translation we get the following mCRL2 specification:

$$\begin{aligned}
\textbf{proc} \quad X^{Chi}_{\text{mCRL2}} = \quad & \sum_{t,t':Time_{\text{H}}} ((\lambda_{\mathsf{time}:\mathcal{S}_{Time},pred_u:Time_{\text{H}}\to\mathbb{B}}(pred_u(\mathsf{time}) \\
& \wedge \mathit{if}(U_G(a), \forall_{x:\mathcal{S}_{Time}} x < \mathsf{time} \Rightarrow \neg(pred_u(x)), \mathit{true}))) \\
& \quad (\pi_{time}(t'), \pi_{time}(t), (\lambda_{x:\mathcal{S}_{Time}} x \geq 1))) \to \\
& get_{\text{time}}(t) \mid a{\cdot}t' \mid \mathit{diff}(\mathsf{time}) \mid set_{\text{time}'}(t');
\end{aligned}$$

The corresponding implementation of the model can be found in Appendix B.2.1.

Discussion The specification contains no model variables. Therefore a memory process and value exchange actions become optional. If we would translate the memory process, we would gain a (global) memory process that can only executes τ actions, thereby introducing τ loops before and after performing action a. Since the memory process is irrelevant, we remove it and obtain a specification for which the behavior is branching bisimilar w.r.t. the translated specification that contains the memory process.

To simulate the model we apply two modifications. The first modification is required to linearize the specification. That is, we need to map the time domain with micro steps to a time domain without a micro steps, since the tools *only* allow the sorts $\mathbb{N}, \mathbb{N}^+, \mathbb{Z}$ or $\mathbb{R}$. Other (strictly) ordered sorts are not supported for the '$\cdot$'-operator. The abstraction can be performed safely, since no two actions occur at the same moment in time. The second modification is required for simulation purposes. Here we change the alias for the $\mathcal{S}_{Time}$ to $\mathbb{N}$, since the simulation tools do not allow enumeration over dense domains (such as $\mathbb{R}$).

After applying the modifications, we can simulate the behavior. If we now change the urgency mapping of $U_G(a)$ to *false*, all time points ≥ 2 become valid. Because $\mathbb{N}$ has no upper bound, the behavior describes an infinite branching structure (Chapter 5.4.3) that cannot be explored exhaustively.

5.5.2 Alternative Composition Example

To illustrate the effect of (non)-urgent action updates, and the translation of the alternative composition we consider the following specification:

$$\begin{aligned}
& \langle \mathsf{time} \geq 2 \to a : \emptyset : \mathit{true} \;[\!]\; 10 \geq \mathsf{time} \geq 1 \to b : \emptyset : s = \mathit{false}, \\
& \{\mathsf{time} \mapsto 0, s \mapsto \mathit{true}\}, (\{\mathsf{time} \mapsto \mathsf{cont}, s \mapsto \mathsf{disc}\}, \{a \mapsto \mathit{true}, b \mapsto \mathit{false}\}, \emptyset, \emptyset)\rangle
\end{aligned}$$

The urgency restricts the length of the delay. Because a is an urgent action and b is a non-urgent action, time can progress until action a is performed. This means that between the time points $1 \leq \mathsf{time} \leq 2$ the process can choose to perform the non-urgent action b. On $\mathsf{time} \approx 2$ it can perform the urgent action a as well. If the process chooses to perform the action b, it sets the value of the variable s to *false*.

To compute the maximal delay, we represent the first argument of the Δ^{max}_{time} (which is the interpretation of a Chi 2.0 process term) into a suitable mCRL2 data expression. Hence we interpret a Chi 2.0 process terminate as the structured sort χ. For the relevant signature, every function symbol of a Chi 2.0 term is mapped to a separate constructor. The corresponding arguments of a term are represented as the arguments of the constructor function. The alternative composition is modeled via the constructor function x_{alt} and the guarded action update is modeled via the constructor function $x_{u \to a:W:r}$:

$$
\begin{array}{ll}
\textbf{sort} & \chi = \textbf{struct} \\
& \quad x_{alt}(\pi_1 : \chi, \pi_2 : \chi) \\
& \quad \mid x_{u \to a:W:r}(\pi_u : \mathbb{B} \times \mathcal{S}_{Time} \to \mathbb{B}, \pi_a : \mathcal{L}_{\text{basic}}, \\
& \qquad\qquad \pi_{diff} : Set(\mathcal{V}), \pi_r : \mathbb{B} \times \mathbb{B} \times \mathcal{S}_{Time} \to \mathbb{B});
\end{array}
$$

We assume that $dom(\pi_u)$ is modeled by $\vec{\mathcal{S}_{\mathcal{V}}} \times \mathcal{S}_{Time}$, where $\vec{\mathcal{S}_{\mathcal{V}}}$ informally states the sorts that are associated to the model variables with their current values. Since all variable are of sort $\mathbb{B}$ and the set of variables consists of, $dom(\pi_u)$ is modeled as $\mathbb{B} \times \mathcal{S}_{Time}$. For $dom(\pi_r)$ we assume that it models $\vec{\mathcal{S}_{\mathcal{V}}} \times \vec{\mathcal{S}_{\mathcal{V}}} \times \mathcal{S}_{Time}$, where the $\vec{\mathcal{S}_{\mathcal{V}}}$ informally states the sorts that are associated to the model variables with their current values. The second $\vec{\mathcal{S}_{\mathcal{V}}}$ states the associated sorts that are to the model variables with their updated values. Since all variable are of sort $\mathbb{B}$ and the set of variables consists of s, $dom(\pi_u)$ is modeled as $\mathbb{B} \times \mathbb{B} \times \mathcal{S}_{Time}$.

To compute Δ^{max}_{time} we specify the following data equations. We have taken the liberty to pre-compute (and ease) the urgency functions for both the actions a and b.

$$
\begin{array}{ll}
\textbf{map} & \Delta^{max}_{time} : \chi \times \mathbb{B} \times \mathcal{S}_{Time} \to \mathcal{S}_{Time}; \\
\textbf{var} & p_1, p_2 : \chi; \\
& t : \mathcal{S}_{Time}; \\
& s : \mathbb{B}; \\
& w : Set(\mathcal{V}); \\
& u : \mathbb{B} \times \mathcal{S}_{Time} \to \mathbb{B}; \\
& r : \mathbb{B} \times \mathbb{B} \times \mathcal{S}_{Time} \to \mathbb{B}; \\
\textbf{eqn} & \Delta^{max}_{time}(x_{alt}(p_1, p_2), t, s) = \min(\Delta^{max}_{time}(p_1, t, s), \Delta^{max}_{time}(p_2, t, s)); \\
& \Delta^{max}_{time}(x_{u \to a:W:r}(u, a, w, r), t, s) = if(U_G(a), 2, \infty); \\
& \Delta^{max}_{time}(x_{u \to a:W:r}(u, b, w, r), t, s) = if(U_G(b), 1, 10);
\end{array}
$$

If we apply the translation, we observe that the modeled guarded action update in the left branch of the alternative composition does not require the value of variable s. To simplify the specification, we model that value by a global variable dc (i.e., a variable with a don't care value). So, the transformation specifies the following mCRL2 process, that corresponds to the model provided in Appendix B.2.2

$$
\begin{array}{ll}
\textbf{glob} & dc : \mathbb{B}; \\
\textbf{proc} & X^{Chi}_{\mathrm{mCRL2}} = \sum_{t:Time_{\mathrm{H}}} \sum_{t':Time_{\mathrm{H}}} \sum_{v_1:\mathbb{B}} \sum_{w_1:\mathbb{B}} \\
& (\pi_{time}(t') - \pi_{time}(t) \leq \\
& \quad \Delta^{max}_{time} \left(\begin{array}{l} x_{alt}(\, x_{u\to a:W:r}(\lambda_{v_1:\mathbb{B},\mathsf{time}:\mathcal{S}_{Time}} \mathsf{time} \geq 2, \\ \qquad a, \emptyset, \lambda_{v_1:\mathbb{B},w_1:\mathbb{B},\mathsf{time}:\mathcal{S}_{Time}} true), \\ \quad x_{u\to a:W:r}(\lambda_{v_1:\mathbb{B},\mathsf{time}:\mathcal{S}_{Time}} \mathsf{time} \leq 10 \wedge \mathsf{time} \geq 1, \\ \qquad b, \{s\}, \lambda_{v_1:\mathbb{B},w_1:\mathbb{B},\mathsf{time}:\mathcal{S}_{Time}} w_1 \approx false)), \\ \qquad v_1, \pi_{time}(t) \end{array} \right) \,) \to \\
& \Gamma_{\{get_{\mathrm{time}}|get_{\mathrm{time}}\to get_{\mathrm{time}}, set_{\mathrm{time}'}|set_{\mathrm{time}'}\to set_{\mathrm{time}'}, get^s_{\mathrm{mem}}|get^s_{\mathrm{mem}}\to get^s_{\mathrm{mem}}, set^s_{\mathrm{mem}'}|set^s_{\mathrm{mem}'}\to set^s_{\mathrm{mem}'}\}} \\
& \left(\begin{array}{l} get_{\mathrm{time}}(t) \mid get_{\mathrm{mem}}(v_1) \mid set_{\mathrm{time}'}(t') \mid \\ \left(\begin{array}{l} \left(\begin{array}{l} \sum_{t,t':Time_{\mathrm{H}}} \\ \left(\begin{array}{l} \lambda_{v_1:\mathbb{B},\mathsf{time}':\mathcal{S}_{Time},\mathsf{time}:\mathcal{S}_{Time},pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \\ \left(\begin{array}{l} pred_u(\mathsf{time}') \wedge \\ if(U_G(a), \\ \quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\ \qquad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1, t'')), \\ \quad true) \end{array} \right) \\ (dc, \pi_{time}(t'), \pi_{time}(t), \lambda_{pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}}(\mathsf{time} \geq 2)) \end{array} \right) \end{array} \right) \\ \quad \to (get_{\mathrm{time}}(t) \mid a \mid diff(\mathsf{time}) \mid set_{\mathrm{time}'}(t') \cdot t') \end{array} \right) \\ + \\ set^s_{\mathrm{mem}'}(w_1) \mid \\ \left(\begin{array}{l} \left(\begin{array}{l} \sum_{w_1:\mathbb{B}} \sum_{t,t':Time_{\mathrm{H}}} \\ \left(\begin{array}{l} \lambda_{v_1:\mathbb{B},\mathsf{time}':\mathcal{S}_{Time},\mathsf{time}:\mathcal{S}_{Time},pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \\ \left(\begin{array}{l} pred_u(\mathsf{time}') \wedge \\ if(U_G(b), \\ \quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\ \qquad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1, t'')), \\ \quad true) \end{array} \right) \\ (dc, \pi_{time}(t'), \pi_{time}(t), \\ \quad \lambda_{pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}}(\mathsf{time} \leq 10 \wedge \mathsf{time} \geq 1)) \\ \wedge \\ \lambda_{v_1:\mathbb{B},w_1:\mathbb{B},\mathsf{time}':\mathcal{S}_{Time} pred_r:\mathbb{B}\times\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \left(pred_r(v_1, w_1, \mathsf{time}')\right) \\ (dc, w_1, \pi_{time}(t'), \lambda_{pred_r:\mathbb{B}\times\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}}(w_1 \approx false)) \end{array} \right) \end{array} \right) \\ \quad \to \left(get_{\mathrm{time}}(t) \mid set^s_{\mathrm{mem}'}(w_1) \mid b \mid diff(\{s, \mathsf{time}\}) \mid set_{\mathrm{time}'}(t') \cdot t' \right) \end{array} \right) \end{array} \right) \cdot t';
\end{array}
$$

Discussion To simulate the mCRL2 specification we perform the same abstractions as in our previous example. Namely, we abstract from the counter and we restrict the dense time domain. Additionally we also set a time bound in the model by substituting ∞ with a (large) number. The mCRL2 tools assume that specifications are in the pCRL format [GPU01]. This format does not allow the use of the parallel operator, the

block operator, the allow operator, the communication operator or the hide operator inside a process. Because the communication operator occurs inside the scope of the sum operator, the specification is not in the required pCRL format. As the nested communication operator *only* eliminates duplicate value exchanges for model variables, it can be eliminated by a syntactic pre-processing step, after which the model can be linearized and simulated.

5.5.3 Parallel Composition Example

The third example shows the transformation of a process with a parallel composition. The example is identical to the previous one, except that the alternative operator has been replaced by a parallel operator.

$$\langle \mathsf{time} \geq 2 \rightarrow a : \emptyset : \mathit{true} \parallel 10 \geq \mathsf{time} \geq 1 \rightarrow b : \emptyset : s = \mathit{false},$$
$$\{\mathsf{time} \mapsto 0, s \mapsto \mathit{true}\}, (\{\mathsf{time} \mapsto \mathsf{cont}, s \mapsto \mathsf{disc}\}, \{a \mapsto \mathit{true}, b \mapsto \mathit{false}\}, \emptyset, \emptyset)\rangle$$

The corresponding mCRL2 process is provided below. The implemented model can be found in Appendix B.2.3.

$$\begin{array}{ll}
\textbf{glob} & dc : \mathbb{B}; \\
\textbf{proc} & X^{Chi}_{\text{mCRL2}} = \\
& \nabla_{\left\{\begin{array}{c} get^s_{\text{mem}} | set^s_{\text{mem}'} | get_{\text{time}} | a | \mathit{diff} | set_{\text{time}'}, get^s_{\text{mem}} | set^s_{\text{mem}'} | get_{\text{time}} | b | \mathit{diff} | set_{\text{time}'}, \\ get^s_{\text{mem}} | set^s_{\text{mem}'} | get_{\text{time}} | \tau_\chi | \mathit{diff} | \mathit{diff} | set_{\text{time}'} \end{array}\right\}} \\
& \quad \Gamma_{\left\{\begin{array}{c} get_{\text{time}} | get_{\text{time}} \rightarrow get_{\text{time}}, set_{\text{time}'} | set_{\text{time}'} \rightarrow set_{\text{time}'}, get^s_{\text{mem}} | get^s_{\text{mem}} \rightarrow get^s_{\text{mem}}, \\ set^s_{\text{mem}'} | set^s_{\text{mem}'} \rightarrow set^s_{\text{mem}'}, \mathit{diff} | \mathit{diff} \rightarrow \mathit{diff} \end{array}\right\}} \\
& \qquad (Z_1 \parallel Z_2)\,;
\end{array}$$

where Z_1 and Z_2 are respectively defined as:

$$\begin{array}{ll}
\textbf{proc} & Z_1 = \sum_{t:Time_{\text{H}}} \sum_{t':Time_{\text{H}}} \\
& \left(\begin{array}{l} \lambda_{v_1:\mathbb{B}, \mathsf{time}':\mathcal{S}_{Time}, \mathsf{time}:\mathcal{S}_{Time}, pred_u:\mathbb{B}\times\mathcal{S}_{Time}\rightarrow\mathbb{B}} \\ \left(\begin{array}{l} pred_u(v_1, \mathsf{time}') \wedge \\ \mathit{if}(U_G(a), \\ \quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\ \qquad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1, t'')), \\ \quad \mathit{true}) \end{array} \right) \\ (dc, \pi_{time}(t'), \pi_{time}(t), \lambda_{\mathbb{B}\times\mathsf{time}:\mathcal{S}_{Time}}(\mathsf{time} \geq 2)) \end{array} \right) \\
& \quad \rightarrow (get_{\text{time}}(t) \mid a \mid \mathit{diff}(\mathsf{time}) \mid set_{\text{time}'}(t') \triangleleft t')\,;
\end{array}$$

$$
\begin{array}{ll}
\textbf{proc} & Z_2 = \sum_{t:Time_{\mathrm{H}}} \sum_{t':Time_{\mathrm{H}}} \sum_{w_1:\mathbb{B}} \\
& \left(\begin{array}{l}
\lambda_{v_1:\mathbb{B},\mathsf{time}':\mathcal{S}_{Time},\mathsf{time}:\mathcal{S}_{Time},pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \\
\quad \left(\begin{array}{l}
pred_u(v_1,\mathsf{time}') \wedge \\
\ if(U_G(b), \\
\quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\
\qquad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1,t'')), \\
\quad true)
\end{array} \right) \\
\quad (dc, \pi_{time}(t'), \pi_{time}(t), \lambda_{\mathsf{time}:\mathcal{S}_{Time}}(\mathsf{time} \leq 10 \wedge \mathsf{time} \geq 1)) \\
\wedge \\
\lambda_{v_1:\mathbb{B},w_1:\mathbb{B},\mathsf{time}':\mathcal{S}_{Time},\mathsf{time}:\mathcal{S}_{Time},pred_u:\mathbb{B}\times\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \left(pred_r(v_1,w_1,\mathsf{time}') \right) \\
\quad (dc, w_1, \pi_{time}(t'), \pi_{time}(t), \lambda_{v_1:\mathbb{B},w_1:\mathbb{B},\mathsf{time}:\mathcal{S}_{Time}}(w_1 \approx false))
\end{array} \right) \\
\quad \to \left(set^s_{\mathrm{mem}'}(w_1) \mid get_{\mathrm{time}}(t) \mid b \mid diff(\{s,\mathsf{time}\}) \mid set_{\mathrm{time}'}(t')^c t' \right);
\end{array}
$$

Discussion The example with the parallel composition is (unlike to the alternative composition example) in pCRL format. Hence it is not required to perform a syntactic pre-processing steps. To linearize the process, we need to abstract from the time domain with micro steps, as we have seen in the previous examples [†].

5.5.4 Communication Example

The fourth example illustrates a value exchange over a communication channel. The sending process sends the negated value of s (which is initially *true*, thereby sending *false*). The receiving end updates the value of s by the negation. The Chi 2.0 model that expresses the behavior is specified as:

$$
\begin{array}{l}
\langle \mathsf{time} \geq 2 \to c!(\neg s) : \emptyset : true \parallel \mathsf{time} \geq 1 \to c?(s) : true, \\
\{\mathsf{time} \mapsto 0, s \mapsto true\}, (\{\mathsf{time} \mapsto \mathsf{cont}, s \mapsto \mathsf{disc}\}, \{c! \mapsto true, c? \mapsto false\}, \emptyset, \emptyset)\rangle
\end{array}
$$

Chapter 5.3.6 states that we define urgency for the communicating channel ends instead of the communicating channels. We here exemplify how this could be achieved. So, we model the sending part of the channel urgent and the receiving part of the channel non-urgent. For presentation purposes we only present the allowed set of multi-actions to those that are executed after a successful communication between channels that are relevant. Although other multi-actions are not shown here, they are present in the implemented model. The model can be found in Appendix B.2.4. According to the transformation we obtain the following mCRL2 process[‡]:

[†]Simulation can only be performed for an untimed linearization (release February 2012). A timed linearization results in an LPS, that requires an infinite amount of variables to find all valuations to satisfy a proposition.

[‡]For presentation purposes we introduce process Z.

$$
\begin{array}{ll}
\textbf{glob} & dc : \mathbb{B}; \\
\textbf{proc} & X^{Chi}_{\mathrm{mCRL2}} = \\
& \nabla \left\{ \begin{array}{c}
\vdots \\
get_{\mathrm{time}} | comm_c | diff | set_{\mathrm{time}'}, get_{\mathrm{time}} | comm_c | diff | diff | set_{\mathrm{time}'}, \\
get^s_{\mathrm{mem}} | get_{\mathrm{time}} | comm_c | diff | set_{\mathrm{time}'}, get^s_{\mathrm{mem}} | get_{\mathrm{time}} | comm_c | diff | diff | set_{\mathrm{time}'}, \\
set^s_{\mathrm{mem}'} | get_{\mathrm{time}} | comm_c | diff | set_{\mathrm{time}'}, set^s_{\mathrm{mem}'} | get_{\mathrm{time}} | comm_c | diff | diff | set_{\mathrm{time}'}, \\
get^s_{\mathrm{mem}} | set^s_{\mathrm{mem}'} | get_{\mathrm{time}} | comm_c | diff | set_{\mathrm{time}'}, get^s_{\mathrm{mem}} | set^s_{\mathrm{mem}'} | get_{\mathrm{time}} | comm_c | diff | diff | set_{\mathrm{time}'}, \\
\vdots
\end{array} \right\} \\
& \Gamma \left\{ \begin{array}{c}
get_{\mathrm{time}} | get_{\mathrm{time}} \to get_{\mathrm{time}}, set_{\mathrm{time}'} | set_{\mathrm{time}'} \to set_{\mathrm{time}'}, send_c | recv_c \to comm_c, \\
get^s_{\mathrm{mem}} | get^s_{\mathrm{mem}} \to get^s_{\mathrm{mem}}, set^s_{\mathrm{mem}'} | set^s_{\mathrm{mem}'} \to set^s_{\mathrm{mem}'}, diff | diff \to diff
\end{array} \right\} \\
& \quad Z; \\
\textbf{proc} & Z = \\
& \quad \sum_{t:Time_{\mathrm{H}}} \sum_{t':Time_{\mathrm{H}}} \sum_{v_1:\mathbb{B}} \\
& \quad \left(\left(\left(\begin{array}{l}
\lambda_{v_1:\mathbb{B}, \mathsf{time}':\mathcal{S}_{Time}, \mathsf{time}:\mathcal{S}_{Time}, pred_u:\mathbb{B}\times\mathcal{S}_{Time}\to\mathbb{B}} \\
\left(\begin{array}{l}
pred_u(v_1, \mathsf{time}') \wedge \\
if(U_G(b), \\
\quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\
\quad\quad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1, t'')), \\
\quad true)
\end{array} \right)
\end{array} \right) \right) \right) \\
& \quad\quad (dc, \pi_{time}(t'), \pi_{time}(t), \lambda_{v_1:\mathbb{B}, \mathsf{time}:\mathcal{S}_{Time}}(\mathsf{time} \geq 2)) \\
& \quad \to \left(get^s_{\mathrm{mem}}(v_1) \mid get_{\mathrm{time}}(t) \mid send_c(\neg v_1) \mid diff(\mathsf{time}) \mid set_{\mathrm{time}'}(t') \lessdot t' \right) \\
& \| \\
& \quad \sum_{t:Time_{\mathrm{H}}} \sum_{t':Time_{\mathrm{H}}} \sum_{w_1:\mathbb{B}} \\
& \quad \left(\left(\left(\begin{array}{l}
\lambda_{\mathsf{time}':\mathcal{S}_{Time}, \mathsf{time}:\mathcal{S}_{Time}, pred_u:\mathcal{S}_{Time}\to\mathbb{B}} \\
\left(\begin{array}{l}
pred_u(v_1, \mathsf{time}') \wedge \\
if(U_G(b), \\
\quad \forall_{t'':\mathcal{S}_{Time}} \mathsf{time} \leq t'' \wedge \\
\quad\quad t'' < \mathsf{time}' \Rightarrow \neg(pred_u(v_1, t'')), \\
\quad true)
\end{array} \right)
\end{array} \right) \right) \right) \\
& \quad\quad (\pi_{time}(t'), \pi_{time}(t), (\lambda_{\mathsf{time}:\mathcal{S}_{Time}} \mathsf{time} \geq 2)) \\
& \quad \to \left(set^s_{\mathrm{mem}'}(w_1) \mid get_{\mathrm{time}}(t) \mid recv_c(w_1) \mid diff(\{s, \mathsf{time}\}) \mid set_{\mathrm{time}'}(t') \lessdot t' \right);
\end{array}
$$

Discussion To linearize the communication example we abstract from the time domain with micro steps, for the same reasons as we have seen in the previous examples. After performing these abstractions it is possible to simulate the model. Observe that the negated value of s is indeed transferred via the communication channel c, and the model variable s is updated accordingly.

5.6 Related Work

The Chi 2.0 language and the mCRL2 language have respectively evolved from the languages χ (0.8, 1.0) [BGR+03, BTW+05] and μCRL [GP93]. Since both of the languages have changed significantly, we felt to reconsider the transformation of [WF05]. An overview on the syntactic and semantic differences between Chi 2.0 and its predecessors is found in [BHR+08]. For mCRL2 and its predecessor, the differences can be found in [GMWU06].

A large amount of the related work has been carried out for the predecessors of the Chi 2.0 language. This includes the work of transformations to the state-based imperative language PROMELA [NO96], to the timed automaton language UPPAAL [BLL+95] and to the process algebra language μCRL [WF05].

Our work mostly resembles the work of [WF05]. The work of Wijs and Fokkink shows a timed translation from Chi (0.8) to μCRL. It differs in two aspects, namely (i) the translation of time, and (ii) the considered kind of specifications.

Ad (i), the original μCRL language includes no native notion of time. The work of [WF05] shows a way to model time by discrete (uniform) time-steps and performs the translation to the extended model. Here, the authors where free to choose their interpretation of time. In mCRL2 the notion of time has been fixed to sorts with a strict total order, for which the sort $\mathbb{R}$ is the only sort allowed by tools. This allows for non-uniform time-steps between any two actions. Unfortunately, the *current* mCRL2 time model is incompatible with the time model within Chi 2.0, as it cannot preserve the order for any two actions at the same moment in time or perform an action at time 0. For that reason, we opt to include a time model with micro steps in the translation.

Ad (ii), our work describes a translation of a Chi 2.0 *specification* to an mCRL2 *specification*. The work of [WF05] translates a *linearized* Chi specification to an LPS. As a result, model variables are translated differently. Our work captures them by separate processes, apart from the translated Chi 2.0 process. In the work of [WF05], they are added as process parameters in the LPS. Hence, we feel that our work is an extension of the work of [WF05], because we translate a larger set of notions. Moreover, we consider the linearization of a processes as a separate task, e.g., as in [Use02], because it can be hard or even impossible to perform the task successfully.

5.7 Conclusions

This chapter shows the denotational translation between two formal specification languages. The source language is the hybrid formalism Chi 2.0, foremost suitable for formal simulation. The selected target formalism is mCRL2, that enables the verification of modal properties for translated models.

Since both languages contain notions that are incompatible, we apply an abstraction to the source language, such that we retain a set of notions that we can transform. We abstract from almost all continuous notions. As already indicated, it is difficult and for some cases even impossible to exactly compute the values for ordinary differen-

tial equations. Although the mCRL2 toolset facilitates a higher order rewrite system, the toolset is currently incapable to solve these kind of equations. Hence, we only translate a subset of the Chi 2.0 language.

Even though parts of the languages are translation-wise compatible, the compositional transformation is still complex. We hint that the correspondence is maintained during the translation. To strengthen the validity of the transformation, we validate the behavior for a limited set of examples. Here, we observe that the behavior from the mCRL2 models is indeed dictated by the Chi 2.0 models. Consequently, if one verifies a property, and wants to assert that the property is preserved in the Chi 2.0 model, the verification results (e.g., the proofs/counterexamples provided by the model-checker) need to be executed by the Chi 2.0 model. Hence, it implies that one has to assume that the complex translation is correct (or provide a proof).

The resulting mCRL2 models are complex. Even for small models, the behavior can be too complex to be linearized, simulated or analyzed by (only) tools. These problems are devoted to the following four concepts.

Firstly, the tools that (currently) support the mCRL2 language are picky with time. This means that tools can only deal with time if it is of sort $\mathbb{R}$. Because we use a time domain with micro steps, we either have to apply an abstraction or a transformation, such that the hybrid domain is mapped to the proper timed domain. Hence we have to apply concessions to the properties that we can verify, alter the model in a post processing step, or state verification properties differently.

Secondly, to mimic the strong time deterministic choice in a weak time deterministic setting we introduce a function that computes the maximal delay between alternatives. If both actions in the alternatives are non-urgent, it is not guaranteed that a maximum exists. If no maximum exists the mCRL2 tools are not able to compute a solution and no simulation can be performed. Therefore we slightly alter the model (e.g., set a time bound on the model) such that we can perform simulations.

Thirdly, when models can be linearized, it provides no guarantee that we can perform a simulation. Especially, if a guard depends on the value of the $\mathsf{time} \in \mathbb{R}$ variable, the simulation and verification tools need to enumerate values over dense domains. Hence, for simulation purposes, we either restrict the time domain to the (non-negative) values of $\mathbb{N}$, or completely abstract from time.

Fourthly, the proposed transformation uses constructs that are not in the pCRL format [GPU01]. Because the mCRL2 tools can only deal with specifications that are in the pCRL format, it is possible that some transformations require a post-processing step (e.g., to eliminate duplicates actions) or cannot be used at all (e.g., if the parallel operator occurs in the recursion of a variable scope).

Despite the limitations the translation is still valuable. It illustrates that if one expects that two languages are suitable for a compositional transformation or defines the semantics in a denotational manner, many complications may arise. This can be observed in the translation, the resulting models, and the absence of a proof for the behavioral preservation. Moreover, this approach is non-reusable. Hence, if the language is subjected to change (i.e., which happened for the CIF [BRSR07], that evolved from the Chi 2.0 language) the entire transformation needs to be reconsidered.

Chapter 6

Disseminating Verification Results

6.1 Introduction

The previous three chapters have shown how to create formal models for different kinds of specifications. These verification models have their own abstractions and design decisions w.r.t. the design models. This implies that when conducting verification, it is often not possible to directly transfer the results back to the design domain. Hence, explaining verification results is perceived to be difficult.

To ease this problem, interdisciplinary modeling methods are required. These modeling methods should integrate formal methods with existing development trajectories. To be successful, these methods should be lightweight to apply and easy to understand, preferably visually. This chapter describes a generic co-design solution that takes the result from a formal analysis and uses it to animate physical designs. To capture the virtual physical design, we use *Computer Aided Design* (CAD) models. This chapter describes an instance of a bridge between the analysis environment and the interchange environment from Figure 1.1.

CAD models allow a precise and a flexible view on the physical characteristics of a system's components. In product development immense costs are associated to physically test them. To reduce the number of expensive tests, it is possible to run virtual tests on CAD models. CAD is also used to virtually present flexible ideas at lower costs and allow error detection (e.g., malformed components, incorrect use of materials) in early development stages. This makes them highly suitable in a development process. The models are also used to simulate objects in a certain (virtual) physical environments. These simulations often consist of predefined scenarios or fixed sets of tests which limit the system's analysis. Such simulations can easily miss unpleasant concurrent behavior, like race conditions or deadlocks.

A formal behavioral model contributes towards a more clear and unambiguous understanding of the concurrent behavior of a system. These models ensure that

the behavior is conform the settled requirements. Unfortunately, they are difficult to understand for engineers that practice other disciplines.

The co-design solution that we propose enriches CAD models with behavioral information, extracted from a formal model, such that the concrete behavior is displayed in a (non-interactive) simulated environment of the actual system. Combining these modeling methods into a co-design solution improves system development in several ways. Firstly, it provides engineers a better insight in the system's behavior, since an animation provides information in a way that is easily perceived by human vision. Secondly, it eases communication between different disciplines, since animations visually pin-point a problem. This avoids the study of the formal models. Thirdly, the co-design solution enables the virtual integration of dynamic components. Here, simulations can be used to inspect a system's integration (e.g., observe that moving components do not collide). Fourth, it enables the virtual execution of a systems behavior. These simulations can contribute to a lower number of faults as the formal models complement the physical models.

The content of the chapter is dissected into two parts. Firstly, Chapter 6.2 describes the general approach and specifies the components that are required to realize the visualization. Secondly, in Chapter 6.3 we illustrate the approach by a case study that describes a wafer dryer facility. Chapter 6.4 describes work that has been performed by others. In Chapter 6.5 we provide our concluding remarks.

6.2 Approach

Visualizing the behavior of a formal model requires five components. The first component is an (observable) trace that needs to be visualized. In practical situations, these traces are obtained from the logging of simulation data, are the result of a witness during model checking, or are specified by test scenarios. The second component consists of a kinematic visualizer. A kinematic visualizer is a system that animates changes for virtual objects, which are accommodated by most of the current 3D modeling and animation packages. Additionally, we require that a kinematic visualizer:

- is capable of importing physical models used in an industrial environment.
- allows the visualization of behavior (movement, scaling, rotation, color changes, etc.).
- contains a scripting language that automates visualization tasks.

The third component consists of a set of (virtual) physical models that need to be imported into the kinematic visualizer. The fourth component is a kinematic language that specifies a mapping between the actions in the formal model and the visualized actions in the kinematic visualizer. The fifth component denotes a kinematic preprocessor that facilitates the generation of the animation statements for the given trace in the kinematic language. The animation statements consist of a series of automated tasks that are imported into the kinematic visualizer. Figure 6.1 depicts

the flow among the components. The following explains each of the aforementioned components.

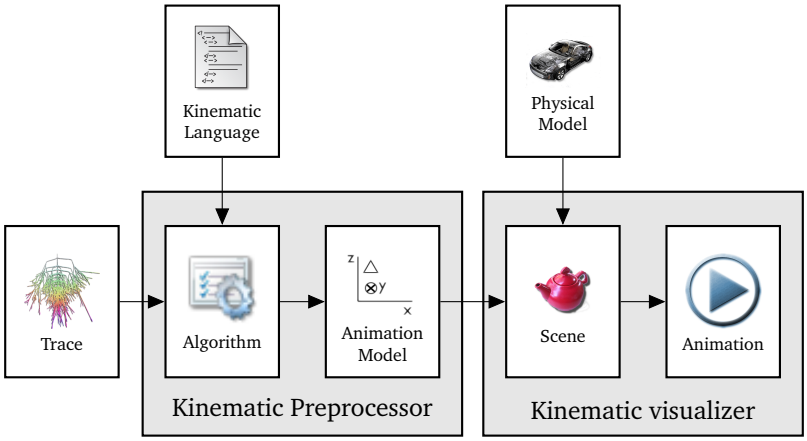


Figure 6.1 Relationship between components for the proposed co-design solution

6.2.1 Action Trace

An *action trace* consists of a vector of timed multi-actions, i.e., $a_T \in \overrightarrow{2^{\mathcal{A}\times\vec{D}\times\mathbb{N}} \times \mathbb{R}}$ where $\mathcal{A}$ represents the set of all action labels, $\vec{D}$ describes the possible data parameters, $\mathbb{N}$ the multiplicity and $\mathbb{R}$ the time at which the action occurs.

The concrete action traces that we consider are described via the following BNF:

$$\begin{array}{lcl} \alpha & ::= & \tau \mid a(\vec{d}) \mid \alpha \mid \alpha \\ P & ::= & \alpha^{\varsigma}t \cdot P \mid \delta^{\varsigma}t \mid \epsilon^{\varsigma}t \end{array}$$

Here, $\alpha^{\varsigma}t$ denotes a multi-action α executed at (absolute) time stamp t. A multi-action α is a collection of actions $a(\vec{d})$ combined by means of $|$, where $a \in \mathcal{A}$ and $\vec{d} \in \vec{D}$. The empty multi-action is denoted by τ, which denotes an internal action. Furthermore, $\cdot$ denotes the concatenation of a timed multi-action with an action trace. A $\delta^{\varsigma}t$ resembles a deadlock or inaction at time stamp t, after which it is impossible to execute any behavior. The empty action trace is denoted by ϵ. We assume that a trace is strictly increasing with respect to the value of the time stamp. For our convenience these traces are a subset of the mCRL2 language.

The relationship between P and a_T is defined through $f : P \rightarrow \overrightarrow{\mathcal{A} \times \vec{D} \times \mathbb{R}}$

$$\begin{array}{lcl} f(\alpha^{\varsigma}t \cdot P) & = & f'(\alpha, t) +\!\!+ f(P) \\ f(\delta^{\varsigma}t) & = & [\langle \delta, [], t \rangle] \\ f(\epsilon^{\varsigma}t) & = & [\langle \epsilon, [], t \rangle] \end{array}$$

and by $f' : \alpha \times \mathbb{R} \rightarrow \overrightarrow{\mathcal{A} \times \vec{D} \times \mathbb{R}}$:

$$\begin{array}{lcl} f'(\tau, t) & = & [\langle \tau, [], t \rangle] \\ f'(a(\vec{d}), t) & = & [\langle a, \vec{d}, t \rangle] \\ f'(a_1 \mid a_2, t) & = & f'(a_1, t) +\!\!+ f'(a_2, t) \end{array}$$

6.2.2 Physical Model

A physical model describes the set of all possible objects that need to be visualized in an animation. An object describes the characteristics of a visual element. The set of objects is represented by *Obj*. *Obj* is split into two disjoint sets Obj_{static} and Obj_{dynamic}. The set of static objects Obj_{static} does not change in an animation. They have a fixed position, rotation, scale and color. The set of dynamic objects Obj_{dynamic} can potentially change their position, rotation, scale and color during an animation.

To visualize an object we require an instance of an object. This means that if we have an object named "tea-pot", and we want to animate two tea-pots, we need to derive two instances. Moreover, we require two different named objects if we want to animate two instances for which one is static and the other is dynamic. An example could be a moving tea-pot, combined with a stationary tea-pot. We assume that all instances are taken from the collection of I_{Obj}.

6.2.3 Kinematic Language

The kinematic language describes the relation between actions of the action trace, the visualization actions and the kinematic effects associated with the visualization actions.

Every action in an action trace is atomic. In the context of behavioral models this implies that whenever an action is started, it cannot be interrupted by another action. An action is guaranteed to completely finish or it is not executed at all.

Actions that are visualized often require a certain amount of time to complete (e.g., moving an object from one location to another). This means that visualization actions are not atomic. For this reason, we require that every visualization action is described by two actions. The first action indicates the start of a visualization action. The second action indicates the end of a visualization action.

The set of visualization actions is denoted by $\mathcal{A}_V$. The function $s_{\text{begin}} \in \mathcal{A} \times \vec{D} \mapsto \mathcal{A}_V$ describes the relation between the actions from a trace and the start of a visualization action. Similarly, the function $s_{\text{end}} \in \mathcal{A} \times \vec{D} \mapsto \mathcal{A}_V$ describes the relation between the actions from a trace and the end of a visualization action. Moreover, we require that the start and the end of an action belonging to the same visualization action (i.e., $\forall_{a \in \mathcal{A} \times \vec{D}}(s_{\text{begin}}(a) = s_{\text{end}}(a))$) are executed in alternating order. This requirement is reasonable, since a physical action (e.g., the rise and set of the sun) can only be performed again after an action has ended.

The kinematic language describes the effects a performed action has on the object instance within an animation. The effects are specified via the function $\mathcal{M}$: $\mathcal{A}_V \mapsto$

$(I_{Obj} \mapsto \mathcal{V})$ which describes for every visualization action the affected instances along with their (basic) kinematic effects as a sextuple:

$$\mathcal{V} = \mathcal{C} \times 2^{\mathcal{D}_M \times \mathbb{N}} \times 2^{\mathcal{D}_R \times \mathbb{N}} \times 2^{\mathcal{D}_S \times \mathbb{N}} \times 2^{\mathcal{D}_{rgb\alpha} \times \mathbb{N}} \times \mathbb{B}$$

A (basic) kinematic effect consists of the following elements:

- creation: $\mathcal{C} = \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^4$ denotes the absolute position, rotation, scale and color,
- movement: $\mathcal{D}_M = \mathbb{R}^3$ denotes the relative change for the *xyz*-position,
- rotation: $\mathcal{D}_R = \mathbb{R}^3$ denotes the relative change for the Euler *xyz*-rotation,
- scaling: $\mathcal{D}_S = \mathbb{R}^3$ denotes the relative change for the *xyz*-scale,
- color: $\mathcal{D}_{rgb\alpha} = \mathbb{R}^4$ denotes the relative change for the *rgbα*-coloring,
- destruction: $\mathbb{B}$ denotes whether or not the object must be destroyed.

Undefined elements are represented by a special value $\perp$. Visualization actions that are unspecified in a mapping have no effect in the visualization, i.e., no animations are performed. Advanced kinematic effects, such as following a certain path or twisting a shape of an instance can be added to $\mathcal{V}$ by extending the sextuple. Hence, the kinematic language acts as an abstraction mechanism: It focuses on the relevant aspects to provide a better insight in the behavior of the system by abstracting from the irrelevant behavior. We assume that a visualization action, that describes a change, is only performed between the initialization and the destruction of an instance of an object. Moreover, we assume that internal actions (i.e., τ) are not visualized.

6.2.4 Kinematic Pre-processor

The kinematic pre-processor computes the information that is required for the kinematic visualizer. This information is computed from the action trace and the kinematic language. To visualize an action trace, we pre-process it to determine the pairs of atomic actions that together form a visualization action. Once we have determined the corresponding pairs, we subsequently determine the time period for a visualization action. The length of a visualization action is specified by the amount of time that passes between two consecutive pair-wise actions. If $a(\vec{d})^{\frown}t$ and $b(\vec{e})^{\frown}t'$ are a pair, then the amount of time that has passed is specified by $|t' - t|$. Moreover, we assume that the animation contains no negative time stamps, the visualization actions start at time stamp > 0, and no visualization actions are started that belong to a visualization actions that are already in progress and have a different duration.

Given an action trace and a kinematic language, the kinematic pre-processor computes a bag of visualization actions $F \in 2^{\mathcal{A}_V \times \mathbb{R} \times \mathbb{R} \times \mathbb{N}}$ where for each $(a_v, t_1, t_2, n) \in F$

- $a_v : \mathcal{A}_V$ describes the visualization action label,

- $t_1 : \mathbb{R}$ describe the time stamp at which the visualization action starts
- $t_2 : \mathbb{R}$ describe the time stamp at which the visualization action ends
- $n : \mathbb{N}$ denotes the number of visualization actions that occur with the same action labels with corresponding begin and end stamps.

Algorithm 2 describes the pre-processing process that transforms an action trace to a set of visualization actions. The algorithm consists of two parts. The first part of the algorithm, lines (1-11), collects the actions that belong to the begin and end of visualization actions. These are respectively denoted by the lists $B \in \overrightarrow{\mathcal{A}_V \times \mathbb{R}}$ and $U \in \overrightarrow{\mathcal{A}_V \times \mathbb{R}}$. The second part of the algorithm, lines (12-21), constructs the visualization actions that are represented by F. The set comprehension C denotes the first end action (if present) that is performed after the begin action that corresponds to the same visualization action. To assert that the end of a visualization action is not used twice, the end action is removed in line (20). In lines (14-19) we compose an element for F. Here, we assume that when a begin action has no corresponding end action, the visualization action ends at the last time that occurs in the trace. Finally, in line (22) the algorithm returns the set of visualization actions.

Algorithm 2 uses the following notations. Let $d \in D_1 \times \ldots \times D_n$ be a data structure with n elements. If we specify $\pi_i(d)$, $(1 \leq i \leq n)$ we get the i-th element (of sort D_i) in the data structure. Furthermore, we define $X \uplus Y$ as $\{(d_1, \ldots, d_{n-1}, n + m) | (d_1, \ldots, d_{n-1}, n) \in X, (d_1, \ldots, d_{n-1}, m) \in Y\}$.

The kinematic visualizer requires the following information for the purpose of the animation:

- the bag of initial positions $f_C(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathcal{C} \times \mathbb{R} \times \mathbb{N}}$ for object instances in the animation on a given moment in time:

$$\begin{aligned} f_C(\mathcal{M}, F) = \; & \{(\iota, \pi_1(\mathcal{M}(\pi_1(f))(\iota)), \pi_2(f), \pi(f)_3) \\ & \quad | \; f \in F, \; \iota \in dom(\mathcal{M}(\pi_1(f))), \pi_1(range(\mathcal{M}(\pi_1(f)))) \neq \bot\} \end{aligned}$$

- the bag of the relative movements $f^M_\Delta(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathcal{D}_M \times \mathbb{R} \times \mathbb{R} \times \mathbb{N}}$ for object instances over a certain period of time:

$$\begin{aligned} f^M_\Delta(\mathcal{M}, F) = \; & \{(\iota, \pi_2(\mathcal{M}(\pi_1(f))(\iota)), \pi_2(f), \pi_3(f) - \pi_2(f), \pi(f)_4) \\ & \quad | \; f \in F, \; \iota \in dom(\mathcal{M}(\pi_1(f))), \pi_2(\mathcal{M}(\pi_1(f))(\iota)) \neq \bot\} \end{aligned}$$

- the bag of the Euler rotations $f^R_\Delta(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathcal{D}_R \times \mathbb{R} \times \mathbb{R} \times \mathbb{N}}$ for object instances over a certain period of time:

$$\begin{aligned} f^R_\Delta(\mathcal{M}, F) = \; & \{(\iota, \pi_3(\mathcal{M}(\pi_1(f))(\iota)), \pi_2(f), \pi_3(f) - \pi_2(f), \pi(f)_4) \\ & \quad | \; f \in F, \; \iota \in dom(\mathcal{M}(\pi_1(f))), \pi_3(\mathcal{M}(\pi_1(f))(\iota)) \neq \bot\} \end{aligned}$$

Algorithm 2 Algorithm to match visualization actions

Require: a_T

1: **while** a_T.getLength() $\neq 0$ **do**
2: $\alpha \leftarrow a_T$.getHead()
3: **if** α.action() $\in dom(s_{\text{begin}})$ **then**
4: $B \leftarrow (\alpha.\text{action}(), \alpha.\text{time}())$ $\triangleright$ B
5: **else if** α.action() $\in dom(s_{\text{end}})$ **then**
6: $U \leftarrow (\alpha.\text{action}(), \alpha.\text{time}())$ $\triangleright$ U
7: **else if** a.action() $= \delta \vee \alpha$.action() $= \epsilon$ **then**
8: $c_{time} \leftarrow \alpha$.time()
9: **end if**
10: a_T.removeHead()
11: **end while**
12: **for all** $b \in B$ **do**
13: $C \leftarrow \{u \mid u \in U \ \wedge s_{\text{begin}}(\pi_1(b)) = s_{\text{end}}(\pi_1(u)) \wedge \pi_2(u) = \text{Min}\{\pi_2(\text{z}) \mid \text{z} \in \text{U} \wedge \pi_1(\text{u}) = \pi_1(\text{z}) \wedge \pi_2(\text{b}) \leq \pi_2(\text{z})\}$
14: **if** $C \neq \emptyset$ **then**
15: Let $c \in C$
16: $F \leftarrow F \uplus \{(s_{\text{begin}}(\pi_1(b)), \pi_2(b), \pi_2(c), 1)\}$
17: **else**
18: $F \leftarrow F \uplus \{(s_{\text{begin}}(\pi_1(b)), \pi_2(b), c_{time}, 1)\}$
19: **end if**
20: $U \leftarrow U - \{c\}$
21: **end for**
22: **return** F

- the bag of the scaling operations $f_\Delta^S(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathcal{D}_S \times \mathbb{R} \times \mathbb{R} \times \mathbb{N}}$ for object instances over a certain period of time:

$$\begin{aligned} f_\Delta^S(\mathcal{M}, F) = \ & \{(\iota, \pi_4(\mathcal{M}(\pi_1(f))(\iota)), \pi_2(f), \pi_3(f) - \pi_2(f), \pi(f)_4) \\ & \mid f \in F,\ \iota \in dom(\mathcal{M}(\pi_1(f))), \pi_4(\mathcal{M}(\pi_1(f))(\iota)) \neq \bot\} \end{aligned}$$

- the bag of the color change operations $f_\Delta^{rgb\alpha}(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathcal{D}_{rgb\alpha} \times \mathbb{R} \times \mathbb{R} \times \mathbb{N}}$ for object instances over a certain period of time:

$$\begin{aligned} f_\Delta^{rgb\alpha}(\mathcal{M}, F) = \ & \{(\iota, \pi_5(\mathcal{M}(\pi_1(f))(\iota)), \pi_2(f), \pi_3(f) - \pi_2(f), \pi(f)_4) \\ & \mid f \in F,\ \iota \in dom(\mathcal{M}(\pi_1(f))), \pi_5(\mathcal{M}(\pi_1(f))(\iota)) \neq \bot\} \end{aligned}$$

- the bag of object instances $f_D(\mathcal{M}, F) \in 2^{I_{Obj} \times \mathbb{R} \times \mathbb{N}}$ that need to be removed from the scene at a given moment in time:

$$f_D(\mathcal{M}, F) = \ \{(\iota, \pi_3(f), \pi_3(f)) \mid f \in F,\ \iota \in dom(\mathcal{M}(\pi_1(f)))\}$$

Each of these bags describe a relative change for an instance in a scene. If action intervals (with same change function for the same object) overlap, (e.g., let $a, b \in f_v$ such that $\pi_2(b) \leq \pi_2(a) \leq \pi_3(b)$ or $\pi_2(b) \leq \pi_3(a) \leq \pi_3(b)$) the sum over the overlapping vectors is taken. The differential changes are constant over the length of the visual actions.

When two atomic actions $a(\vec{d})$ and $b(\vec{e})$ occur in the same multi-action and together they form a visualization action, i.e., $s_{\text{begin}}(a(\vec{d})) = s_{\text{end}}(b(\vec{e}))$, then the kinematic effect of this visualization action is visualized as a discrete (instant) change.

Furthermore we assume that within the kinematic visualizer, object instances are not destroyed before being created. We also assume that object instances are only animated between the time stamps where an object instance is created and destroyed.

6.2.5 Kinematic Visualizer

To animate the physical behavior we use the kinematic visualizer. The kinematic visualizer contains a virtual environment, wherein all relevant CAD objects are assembled. Such a virtual environment is called a *scene*. The scene is used to synthesize an animation.

Before we animate the physical behavior, all object instances are merged into a scene. All instances that belong to Obj_{static} get a fixed position, rotation, scale, and color. All instances that belong to Obj_{dynamic} are guided by the visualization actions. This means that all actions from $F_C(\mathcal{M}, F)$, $F_\Delta^M(\mathcal{M}, F)$, $F_\Delta^R(\mathcal{M}, F)$, $F_\Delta^S(\mathcal{M}, F)$, $F_\Delta^{rgb\alpha}(\mathcal{M}, F)$ and $F_D(\mathcal{M}, F)$ are assigned to the corresponding instances.

In general, current state-of-the-art 3D modeling and animation software packages such as Autodesk® Maya®, Autodesk® 3D studio Max®, Blender Foundation's Blender, NewTek Lightwave™, etc ..., can import object instances into a scene. To assign the visual effects to the different instances, we require a scripting language.

Once all object instances are merged and the visualization actions are assigned, the trace can be animated. Real-time visualizations can be used to quickly analyze the problem, but often carry less (geometric) detail by providing an instant view. Non-real time visualizations can be used for presentation and demonstration purposes. Together with light emitting sources and various bitmaps for materials, photo realistic animations can be generated.

6.3 Case Study

To study the applicability for this approach we have taken a small industrial case study. The case study describes a wafer handler drying facility to be used in a wafer printing device. The handler must dry individual wafers for at least sixty seconds. The schematic control flow for the wafers of the drying facility is illustrated in Figure 6.2.

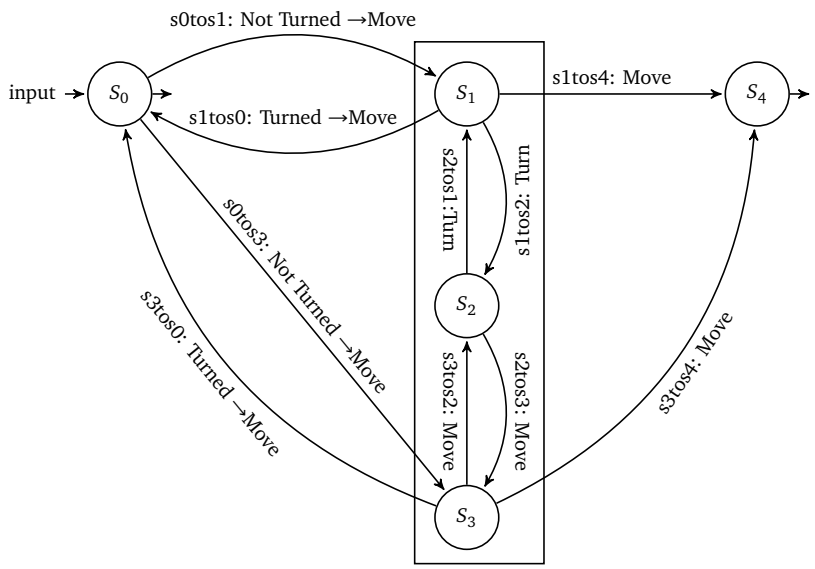


Figure 6.2 Schematic control flow for the wafer drying facility

The handler has a vertical column that offers three slots to position wafers. These slots are numbered S_1, S_2 and S_3, respectively. Slots S_1 and S_2 in the column can switch wafers simultaneously by rotating them upside down. A rotation (or turn movement) takes 5 seconds to complete. A wafer moves from S_2 to S_3, or from S_3 to S_2 in a pan-wise movement. These actions take 3 seconds to complete. Wafers enter the dryer facility unturned via S_0 and are positioned by a non-deterministic choice in the empty slots S_1 or S_3. Wafers enter the dryer with a rate of exactly one wafer every 30 seconds. The system must always accept incoming wafers.

Wafers may only depart the system when they have resided in the dry column for a minimum of 60 seconds. The wafers can depart the system via S_0 if they are in slot S_1 or slot S_3 and are turned. They may also depart the system via S_4 if they are in slot S_1 or slot S_3 in either a turned or unturned position. Moving a wafer in and out of the column takes 3 seconds. The amount of time needed for these movements counts as time that a wafer resides in the drying facility.

All moves and turns are mutually exclusively executed in the system. It is impossible to perform a move or turn when another move or turn is executed. If the slots S_1 and S_2 are filled, a turn is executed as a multi-action of two turns, i.e., the two wafers are rotated and swapped simultaneously.

6.3.1 Design Rules and Assumptions

The controller is modeled under the assumption that the system operates without faults or abnormalities (i.e., the loss of wafers). In the initial state all slots are empty and no wafers are in the dryer facility. For simplicity we only focus on the behavior that is executed by the controller. This implies that the behavioral model does not capture any of the physical properties like the material's stiffness, temperature of the wafers or the humidity in the dryer facility.

The controller stores the kinematic wafer information. The kinematic information is for each wafer linked to an identifier $Id \in \mathbb{N}$, for which the positions are indicated using $Place \in \{S_0, S_1, S_2, S_3, S_4\}$, its turn-status $State \in \mathbb{B}$ (*true* for turned, *false* for unturned), and time stamp that marks the arrival at the system $Stamp \in \mathbb{R}$. Each slot stores at most one wafer. So, there are at most five wafers in the system at all times. Therefore we need at most five identifiers, since the available identifiers can be issued for reuse.

The controller sends different commands to the handlers for moving and rotating the wafer in the dryer facility. We assume that the controller C executes the commands non-deterministically for the wafers that are in the dryer facility DS. This includes the behavior associated to the departure (when a wafer has spent enough time in the dryer facility) and the movements of the wafers between the different slots. The controller does not perform any movement or departure actions when there are no wafers in the system. The entrance of a wafer is scheduled in such a way that the dryer system always accepts incoming wafers, unless all positions that store incoming wafers are occupied. The arrival of wafers is also modeled by the controller. It abstracts from the implementation choose to only model the arrival of the wafers. The model starts at time 1. Commands are sent by the controller to the dryer facility at a fixed rate of at most one command per second.

The behavior of the system is modeled in the mCRL2 language. The (original) model that specifies the system is provided in Appendix B.3.

6.3.2 The Trace

Given the assumptions and modeling decisions, we want to verify that the system cannot deadlock, i.e., the system cannot come into a state from which it cannot perform any actions. It turns out that this property is not satisfied. In Figure 6.3 we depict the associated state space of the behavior the model. We see that the model has a deadlock, denoted by a red dot.

With the help of the mCRL2 toolset, we obtain a trace to the deadlock state. The trace is derived during the explicated state space generation. In fact, there are many traces that lead to the deadlock state. If we inspect the traces, we observe that the actions that are executed, belong to the transport of wafers. So, it makes sense to visualize these actions. The trace that is visualized consists of the actions in the following order:

$$S_0toS_1\_begin(1)^{<}1 \cdot S_0toS_1\_end(1)^{<}4 \cdot S_1toS_2\_begin(1)^{<}19 \cdot S_1toS_2\_end(1)^{<}24 \cdot$$

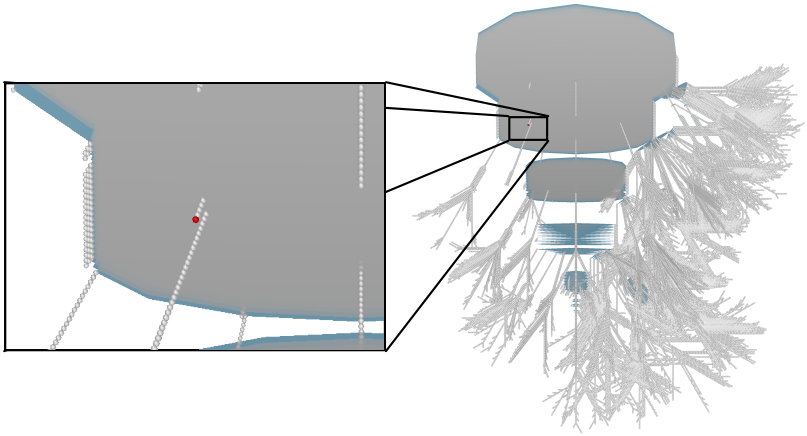


Figure 6.3 The state space for the dryer system with a red colored deadlock

$$S_2toS_1\_begin(1)^{<}26 \cdot S_2toS_1\_end(1) \mid S_0toS_3\_begin(2)^{<}31 \cdot S_0toS_3\_end(2) \mid S_3toS_2\_begin(2)^{<}34 \cdot S_3toS_2\_end(2) \mid S_1toS_2\_begin(1) \mid S_2toS_1\_begin(2)^{<}37 \cdot S_1toS_2\_end(2) \mid S_2toS_1\_end(2) \mid S_2toS_3\_begin(1)^{<}42 \cdot S_2toS_3\_end(1)^{<}45 \cdot \delta^{<}61$$

6.3.3 The Physical Model

For the physical model we merge all object instances into a scene. We import the wafers and a representation of a dryer facility. To transport wafers between positions we choose to abstract from the transport handlers, by making them invisible to the observer.

A wafer's motion between slots describes a non-linear movement, i.e., wafers follow a Bézier-like curve. From an aesthetic point of view, it is nice to define movements that describe a complex movement over time. These movements could have been described by altering $\mathcal{D}_M$, such that it either describes a differential equation or is divided into smaller vectors that need to be chained together. Solving a differential equation is often too complex for kinematic visualizers, because they do not have the solving capabilities. Slicing complex trajectories into smaller linear parts is an intensive time consuming task.

To describe the transport of wafers between the slots, we draw *motionpath*s that the wafers need to follow. A motionpath is a curve (i.e., path) that dictates the movement for an instance of an object over a period of time. A motionpath needs to be specified in the scene in advance, requires a direction and needs to have a unique name. If an object is attached to a motionpath, it moves along the path.

The parts needed for the physical model are depicted in Figure 6.4. The CAD models consist of a dryer system, the wafers and motionpaths.

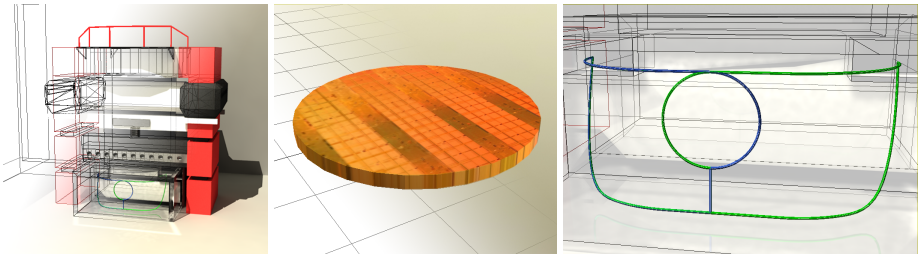


Figure 6.4 Three objects from the physical model

6.3.4 The Interconnecting Model

The interconnecting model is a model written in the kinematic language. Recall that it describes the relation between the performed actions by the trace and the visualized actions. The syntax of this model is described using an XML notation. Before we visualize a trace, we first define the relevant visualization actions. In our case study, all actions correspond to the wafer movements from one slot to another. This concerns the visualization actions `S0toS1(wid)`, `S_0toS3(wid)`, `S_1toS2(wid)`, `S1toS4(wid)`, `S2toS1(wid)`, `S2toS3(wid)`, `S3toS0(wid)`, `S3toS2` and `S3toS4(wid)`, where `wid` denotes the wafer identifier number, i.e., `wid`$\in \mathbb{N}$. The action `Create-S0(wid)` and the action `Destroy-S4(wid)` are important for the visualization, as they respectively denote the creation (entrance in a scene) and the destruction (exit from a scene) for a wafer instance `wid`.

Action Relations

Actions in a trace are defined as atomic discrete event actions. Their corresponding visual behavior is defined for some period of time. So, we need to define which actions from an action trace form a pair in the visual action. This is accomplished by defining an *action relation* between the visualization action and the begin and end actions of the action trace. An action relation `ActionRelation` is defined by:

- The value of the XML element `Action` corresponds to an element of the visualization actions A_V.
- The value of the XML element `BeginAction` corresponds to an element of the begin actions $\mathcal{A} \times \vec{D}$.
- The value of the XML element `EndAction` corresponds to an element from the end actions $\mathcal{A} \times \vec{D}$.

Example 6.1(Action Relation). This example shows how to map s_{begin} and s_{end} respectively to visualization actions with the help of the action relation. We define the

action relation for the create, the destroy and the transport of a wafer from slot S_0 to slot S_1. The action relation for the create of wafer with `wid` $= 1$ is described by the first `ActionRelation`. The action relation for the transport of the wafer with `wid` $= 1$ is described by the second `ActionRelation`. The action relation for the destroy of the wafer with `wid` $= 1$ is described by the third `ActionRelation`.

```
<ActionRelation>
  <Action>       Create-S0(1)    </Action>
  <BeginAction>  S0toS1_begin(1) </BeginAction>
  <EndAction>    S0toS1_begin(1) </EndAction>
</ActionRelation>

<ActionRelation>
  <Action>       Destroy-S4(1) </Action>
  <BeginAction>  S3toS4_end(1) </BeginAction>
  <EndAction>    S3toS4_end(1) </EndAction>
</ActionRelation>

<ActionRelation>
  <Action>       S0toS1(1)       </Action>
  <BeginAction>  S0toS1_begin(1) </BeginAction>
  <EndAction>    S0toS1_end(1)   </EndAction>
</ActionRelation>
```

△

Create Action Map

An *action map* element describes the semantics for a visualization action. That is, it specifies whether an action creates a new object instance, changes information on an object instance or destructs an existing object instance. If a visualization action creates a new object (when the value of a `Type` element equals `Create`) it is required to define:

- the initial *xyz*-position (defined by the `Position` element),
- the *xyz* Euler rotation (defined by the `Rotation` element),
- the *xyz*-scale (defined by the `Scale` element), and
- the *rgb*-color value with α opacity channel (defined by the `RGBA` element).

The value of an `Action` element defines the visualization action that creates an object.

Example 6.2(Create Action Map). The create action map creates an object instance. The value of an `Instance` element defines the dynamic instance of an object. The `Action` element defines the label of an action that affects the instance. This example shows the constructor function for an object that corresponds to `wafer(1)` (wafer with `wid` $= 1$). The initial position of the object is equal to the origin of the Euclidean space. The instance of the object is not rotated, nor scaled in any dimension and has a green non-transparent color.

```
<ActionMap>
  <Action>     Create-S0(1)                        </Action>
  <Instance>   wafer(1)                            </Instance>
  <Type>       Create                              </Type>
  <Position>   <X>0</X> <Y>0</Y> <Z>0</Z>          </Position>
  <Rotation>   <X>0</X> <Y>0</Y> <Z>0</Z>          </Rotation>
  <Scale>      <X>1</X> <Y>1</Y> <Z>1</Z>          </Scale>
  <RGBA>       <R>0</R> <G>1</G> <B>0</B> <A>1</A> </RGBA>
</ActionMap>
```

△

Change Action Map

For action maps that describe changes (i.e., the value of the `Type` element equals `Change`), only the relevant elements are specified. As the complex movements are described using motionpaths, we here exchange the `Position` element (denoting the offset movement) with `Motionpath` (stating that it should follow a motionpath).

Example 6.3(Change Action Map). In the change action map, we show the movement for object instance `wafer(1)` by changing its position if `S1toS2(1)` is executed. In this case the instance rotates 180° clockwise orthogonal over the y-axis during the length of a visualization action. While rotating, the object follows motionpath `S1ToS2path`.

```
<ActionMap>
  <Action>     S1toS2(1)                     </Action>
  <Instance>   wafer(1)                      </Instance>
  <Type>       Change                        </Type>
  <Motionpath> S1ToS2path                    </Motionpath>
  <Rotation>   <X>0</X> <Y>-180</Y> <Z>0</Z> </Rotation>
</ActionMap>
```

△

Destroy Action Map

The destroy action map defines the destroy of an existing object instance. This means that the instance is removed from the scene. Hence, the value of an `Type` element must equal `Destruct`. The XML elements that are mandatory are the visualization action, the object and type of the mapping.

Example 6.4(Destroy Action Map). This example shows the destroy action map for the object instance `wafer(1)`. When the action `Destroy-S4(1)` is executed, it performs a destroy.

```
<ActionMap>
  <Action>    Destroy-S4(1)   </Action>
  <Instance>  wafer(1)        </Instance>
  <Type>      Destruct        </Type>
</ActionMap>
```

△

Time Conversion Unit

Apart from action maps, we also require a time conversion unit. The time conversion unit specifies the relation between the time in the visualization actions and the time in the trace.

Defining such a relation is practical, since for high speed processes it acts as a slow-motion function, as it might reveal footage that is missed by the human eye. Moreover, for slow processes it could act as a fast forward function.

By defining the `TimeUnit` the conversion time unit is set. The relation is expressed in frames per second per time unit of the trace.

Example 6.5(Frame Rate). If a time unit in a trace equals 25 frames in a visualization, the value for the `TimeUnit` needs to be set to `25f`:

```
<TimeUnit> 25f </TimeUnit>
```

△

6.3.5 Visualization

In our approach we have chosen to visualize the trace by using the modeling, animation and rendering package Autodesk® 3D studio Max® (2009) [Aut]. This package has been chosen, as (i) it supports a wide range of CAD models and (ii) repetitive tasks can be executed using MaxScript.

After generating an animation, it turns out that the system deadlocks when both S_1 and S_3 are occupied and a wafer enters the drying facility. Because wafers are not enforced to leave the system the deadlock occurs. This implies that the controller should schedule the wafer's departure from the system as soon as possible.

A part of the trace is visualized in Figure 6.5.

6.4 Related Work

Visualizing simulations are often performed ad-hoc, carried out by a special team of engineers, which are often not part of the development trajectory. The resulting simulations are therefore often custom made, not suitable for reuse, are led by an artistic inspirations, or are targeted to marketing and sales. The work presented here, does not focus on these custom made visualizations, but discusses the relation to work of other that try to visually study the behavior of formal models.

The authors of [BHBM07] study the behavior of a printer design using Happy Flow. Happy Flow describes and visualizes the desired path of a print sheet and the ideal movements of parts that influence the paper's motion. Happy Flow uses a kinematic view, where non-idealistic behavior such as friction, jerk of motors and hysteresis are not taken into account. As such, Happy Flow provides a quick design space exploration with respect to job scheduling. The prerequisite of a Happy Flow model is a mechanical layout of a paper path including the position of pinches, switches and

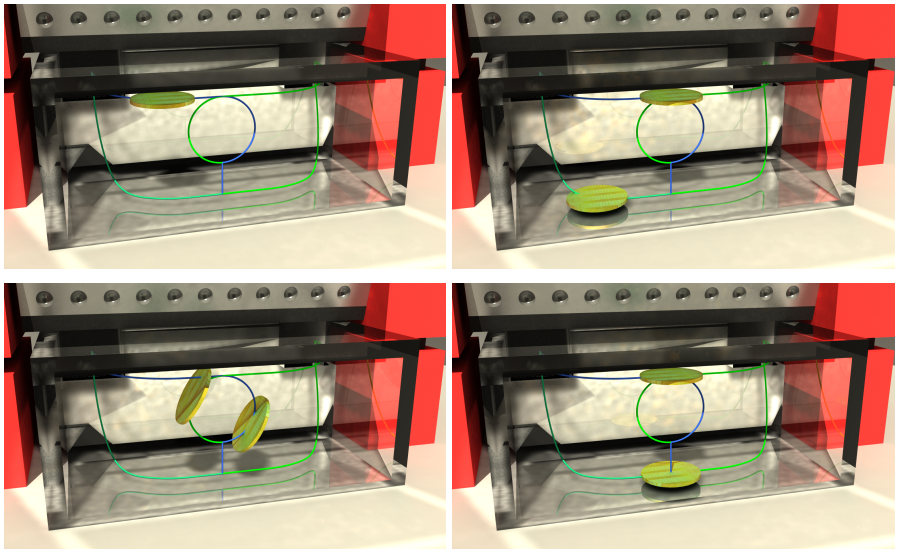


Figure 6.5 Four still images taken from the kinematic visualization

sensors. The mechanical layout is a 2D profile of the machine. The paper path that the paper sheets have to follow, consists of a 1D concatenation of all the registration points that they have to pass, together with their traveling distance and traveling time. The logistics and timing information are combined into position-time diagrams. Although the approach is very effective for the paper document processing industry, it is very difficult to apply it on systems that show other kinds of behavior. The simulation tool requires two dimensional CAD models. In practice it is common to design hardware components in 3D. Therefore whenever a 2D image is needed either a profile is required (which needs to be derived from the 3D model) or custom made images need to be drawn.

Another technique that studies the concurrent behavior of systems are found at the analysis of (large) state space transition graphs. Work of [HWW01, PT08] tries to visualize the behavior of large systems by positioning individual states in an interactive 3D grid space environment. Work of [PW07] focuses on the structure of a graph, but uses a clustering technique based on state attributes and visualizes behavior with state based diagrams. These visualization environments allow simulations by executing actions at an abstract level, which are visualized as transitions between states. The essential difference between the work [HWW01, PW07, PT08] and the work presented in this chapter, is that we generate a concrete visualization (i.e., an animation) instead of providing an abstract visualization.

The authors of [PW06] visualize state transition graphs and extend the visualization with custom diagrams suitable for animation. Here, we see a couple of similarities. Both methods use a formal behavioral model and use a visualization tailored

towards the system under study. The differences between the methods are found in e.g., the kind of models that are used: we use the physical designs of development process, whereas the authors need to draw custom designs. Another difference is that the visualization of [PW06] is based on the state of the model, while our method uses the transitions between states.

6.5 Conclusions

For systems that are built in multi-disciplinary environments, it is crucial to detect faults early in a design phase. As different engineers concurrently develop a system, it is often difficult to explain to other engineers the traces or counter-examples that violate a property. To lower these efforts, and inspired by visual techniques to conduct system analysis, we implemented a general and reusable bridge between formal behavioral models and industrial practice, that with the help of visualization techniques can address shortcomings in system design.

The work presented here originates from a feasibility study where a sketch of the system, a description of the controller's behavior and a set of requirements imposed on the behavior had been given. The analysis of the system has been carried out prior to the development of the system. Therefore all involved engineering disciplines should be able to understand the verification results on the initial designs. Hence, we added a visualization layer for a perceptual dissemination of the verification results, thereby contributing to a better understanding of the system's behavior. By analyzing the behavior prior to the actual development, valuable development resources have been saved.

Moreover, a visual approach enables more easily the sharing of information between developers from different disciplines. The method can be applied in many fields, because the mapping is flexible, relatively easy to deploy, and can be adjusted to the needs of the analysis. To setup an animation, one requires CAD models and formal models. CAD models are often present, since they are required for various reasons. Creating a formal behavioral model requires resources. However it saves expensive test time, because model-checking addresses difficult to detected behavioral characteristics (e.g., race-conditions, rarely occurring deadlocks, etc...) that would probably have been missed by conducting expensive tests. Once the kinematic language is implemented, animations are generated without addressing additional resources. This makes the method also suitable for demonstration purposes.

Worthwhile to mention is that the approach can be used with different kinematic visualizers. The choice of a visualizer depends on the level of expertise of an engineer, and the (dis)advantages of the visualizer. Moreover, the approach can also be used to generate animations for other types of (generated) data. This includes logs from actual system runs or simulations performed by other (formal) behavioral specifications.

Part II

Semantically Engineered Models

Chapter 7

Formalizing a Behavioral Language

7.1 Introduction

The second part of the thesis describes a generic transformation for models of a formal language to a formalism suitable for verification purposes. The approach is explained by four chapters. Since many languages are still informally defined, the first chapter illustrates the formalization of an informal DSL. The second chapter presents the transformation of the formal semantics into a formalism that facilitates automated analysis. The third chapter demonstrates the applicability of the approach for a concrete formal language, namely the mCRL2 language. The fourth chapter reflects on the modeling approach.

To cope with the complexity of control software, model-driven software engineering (MDSE) techniques have widely been adopted over the last decade. MDSE treats models as first class entities and aims at a reduction in the lead-time and a decrease in the development effort, while improving the system's quality in comparison to the more traditional software engineering techniques. Model specifications are commonly created using domain specific languages (DSLs). DSLs are languages that are geared to a well-defined class of problem domains or a particular set of domain aspects. Therefore a DSL has a focused expressiveness as it is targeted towards the jargon [Kle09] to address the problem domain or specific aspects. The second part of the thesis concentrates on executable DSLs, which are used to describe system behavior. These DSLs are mainly used to generate executable models or derive code from concrete domain models. In practice, this means that the semantics of a DSL is implicitly and informally defined in the engine/interpreter that processes the concrete domain models. Occasionally, the formal semantics is stated in a separate chapter, book or document. For languages that are defined in this way, it is difficult to enable the au-

tomated reasoning on the executed behavior. To effectively analyze particular models in these languages, we require an explicit formal semantics, that can, in a processable way, be transformed to a formalism, accommodated with sufficient tool support for formal reasoning.

This chapter concentrates on the first step that is required before we deploy the semantic bridge. It starts with the formalization of an existing, industrial DSL called TRECS [Nie04], for which the execution semantics is defined informally and implicitly through an interpreter. The DSL supports the definition of predictive and reactive rules to optimally allocate manufacturing activities (i.e., tasks) to mechatronic subsystems (i.e., resources) over time while they are subjected to dynamic constraints. Since its conception, the DSL has been tried in an industrial setting where UML-like activity diagrams have been extended with non-disclosed reactive concepts, constraints and run-time optimization rules.

The purpose of the formalization is to capture the intended semantics of the language formally, and compare the result with the actual implemented behavior. We show that formalizing a DSL is likely to uncover sub-optimal design decisions and ambiguities. Thereby, we illustrate how to minimize the formalization impact while retaining backward compatibility. During the formalization both the structure (and relations) of the abstract notions and their behavioral effects are considered. We use Structural Operational Semantics (SOS) [Plo04] to define the effect of language terms from an operational perspective.

Although we formalized almost the entire industrial DSL, this chapter is limited to only the disclosed parts, illustrated by making an Italian dessert: Tiramisu [Sax94]. The preparation of the dessert serves as the running example for this chapter. To focus on relevant aspects, we first make the decisions behind the concrete syntax explicit by projecting it onto an abstract syntax. The projection defines the operators and process variables, which are used to define the formal semantics. The formal semantics is then validated with domain experts and language engineers.

Chapter 7.2 describes the domain related abstract notations and introduces the leading case study. Chapter 7.3 assigns the semantics to the abstract notions and validates that the assigned formal semantics is correct. Chapter 7.4 discusses related work and Chapter 7.5 presents our conclusions.

7.2 Formalizing Domain Notions

To illustrate and discuss all *disclosed features* of the DSL we consider a concrete model that contains the cooking instructions for preparing Tiramisu. The details of the example are provided in Chapter 7.2.1. Chapter 7.2.2 projects the example's concrete syntax onto a formal abstract syntax. Chapter 7.2.3 presents the derived formal abstract notions and taxonomy, after which Chapter 7.2.4 validates that the projection is sound. Chapter 7.2.5 describes the notions that are additionally required to preserve the backward compatibility with the informal language.

ingredient	cream topping	coffee syrup	assembling
milk	500 ml		
white sugar	150 gram	75 gram	
flour	35 gram		
egg yolks	6 pcs.		
dark rum	60 ml	60 ml	
vanilla extract	2 teasp.		
mascarpone cheese	225 gram		
ladyfingers			32 pcs.
espresso		360 ml	
butter	60 gr		

A. Make Tiramisu

1. **Make Cream Topping** - see **B**
2. **add mascarpone** - using a wooden spoon, beat 225 gram mascarpone cheese in a bowl until it is soft and smooth. Then gently whisk the mascarpone into the result from A.1 until the custard mixture is smooth.
3. **mix coffee syrup** - in a large shallow bowl combine 360 ml espresso, 75 grams sugar and 60 ml dark rum.
4. **line loaf pan** - before making layers, take a loaf pan and line it with plastic wrap and making sure that the plastic wrap extends outside the loaf pan to allow wrapping.
5. **Make Layers** - see **C**
6. **cool down cake** - once all layers are made, cover the Tiramisu with plastic wrap and place it in a refrigerator and have it cool for at least 6 hours.
7. **present and serve** - once the cake is cooled, remove the plastic wrap from the top and gently invert the Tiramisu from the loaf pan to a serving plate. Remove remaining plastic wrap and serve the dessert.

B. Make Cream Topping

1. **boil sweet milk** - in a saucepan heat 430 ml milk and 100 grams sugar right up to the boiling point.
2. **egg yolk mixture** - meanwhile whisk 70 ml milk, 50 grams sugar, 35 grams flour and 6 egg yolks in a heatproof bowl.
3. **whisk milk & egg yolk** - once the sweet milk has just come to a boil gradually whisk it into the egg yolk mixture. Transfer this mixture into a large saucepan.
4. **reduce mixture** - next slowly cook the result from B.3 while stirring constantly until it comes to a boil. Once it boils, continue to whisk the mixture constantly for another minute and let it reduce a bit.
5. **enrich mixture** - take the result from B.4 of the heat and strain into a large bowl. Whisk in 60 ml dark rum, 2 teaspoons vanilla extract, and 60 grams butter. Cover the bowl with plastic wrap to prevent crust-forming.
6. **cool down topping** - place the bowl in the refrigerator and let it cool down for approximately two hours.

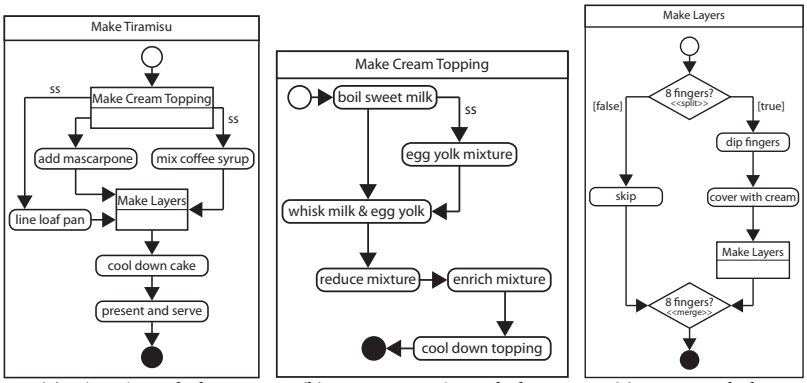
C. Make Layers

1. **8 fingers?** - ensure that we have 8 ladyfingers to create a layer.
2. **dip fingers** - one ladyfinger at a time, dip 8 ladyfingers into the coffee syrup from step A.3 and place them side by side in the loaf pan.
3. **cover with cream** - spoon 1/4 of the custard from A.2 and completely cover the 8 ladyfingers.
4. **make Layers** - repeat **Make Layers** until no more ladyfingers are left.

7.2.1 Running Example

Tiramisu requires ingredients, kitchen utensils and appliances, a recipe and a way of working as specified on Page 131. In the DSL, ingredients, kitchen utensils and appliances are called “resources”. The recipe and its sub-recipes are called “subplans”. The individual activities in a subplan are called “tasks”.

The DSL’s concrete syntax is inspired on activity diagrams, for which the Tiramisu recipe is illustrated in Figure 7.1. Figure 7.1(d) shows the resource definition through a hash-like table. An initialization is required to execute a concrete model. We assume that Make Tiramisu is the initialization and is executed only once. Furthermore we assume that all of the required resources are exactly those which are specified. If we obey these rules, it results in an initialization of ingredients, kitchen utensils and appliances as illustrated in Figure 7.1(e).



(a) Tiramisu subplan (b) Cream Topping subplan (c) Layers subplan

task	consumes	produces
add mascarpone	wooden spoon: 1 pcs mascarpone: 225 gr cooled topping: 1 pcs bowl: 1 pcs	wooden spoon: 1 pcs bowl : 1 pcs custard: 4 vol.
⋮	⋮	⋮

(d) Resource usage

resource(s)	init
milk	480 ml
sugar	225 gr
wooden spoon	1pcs
refrigerator	1 pcs
⋮	⋮

(e) Resource initialization

Figure 7.1 Partial Tiramisu recipe in the DSL’s concrete syntax

7.2.2 Concrete Syntax Projection

Before we assign semantics to syntax, we first define the language’s syntactic notions. We start by identifying and projecting the most elementary notions in terms of behavior to obtain a compositional formal syntax. If we cannot capture the intended behavior by any of the already introduced notions, we either add or refine existing

notions. Note, that we choose a process algebra-like notation like [BBR09], for which we reuse process algebraic operators where possible.

Task

A task is the smallest identifiable behavior in a system under control. In Figure 7.1(a), add mascarpone and mix coffee syrup are tasks. The concrete syntax of a task is denoted by a rounded rectangle (node) with a name label.

Decision 7.1: The execution of a labeled task is atomic and observable. ⌟

Decision 7.2: In some cases we do not want to observe behavior. An example is shown in Figure 7.1(c) by a task labeled with the reserved word skip. This word is represented by a process term τ, denoting an internal non-observable action. For now, $\mathcal{T}$ denotes the finite set of all tasks including τ. ⌟

Precedence Relation

The order of the execution of tasks is restricted by precedence relations. The DSL has two kinds of precedence relations. The first relation describes a finish-start precedence relation (fs) that can be used to start behavior if and only if the preceding behavior has terminated successfully. This is shown in Figure 7.1(a) as a transition without a label between the tasks cool down cake and present and serve. Such a transition informally denotes that the task present and serve may only be executed after the task cool down cake has been successfully (and fully) completed.

The second precedence relation is the start-start precedence relation (ss). This relation starts behavior if and only if the preceding behavior has started its execution. In Figure 7.1(b) such a relation is depicted as a directed edge with label ss between boil sweet milk and egg yolk mixture. Informally, such a transition denotes that the task egg yolk mixture can be started once task boil sweet milk is started, but has not necessarily been fully completed.

Decision 7.3: For both relations we introduce a dedicated operator. Let p and q be process terms. Then the finish-start relation in the abstract syntax is expressed by the finish-start operator $\cdot$ as:

$$p \cdot q$$

The start-start relation in the abstract syntax is expressed by the start-start operator $|||$ as:

$$p \mathbin{|||} q$$

⌟

Decision 7.4: This decision extends Decision 7.1 where all tasks are considered atomic. Atomicity implies that if p would be a single task t, we could not distinguish the behavior of the $|||$ operator from that of the $\cdot$ operator. To make this observable, we introduce for all labeled tasks except τ, an explicit start and finish action. Let $\mathcal{T}_\alpha$ be a

finite set of elements called starting tasks, i.e., the alphabet of starting tasks. Let $\mathcal{T}_\omega$ be a finite set of elements called finishing tasks, i.e., the alphabet of finishing tasks. We refine the labeled tasks such that $t_\alpha \in \mathcal{T}_\alpha$ and $t_\omega \in \mathcal{T}_\omega$ denote the start and the finish of task t respectively. For now, we assume that every action t_α that is performed, t_ω will always eventually follow (Chapter 7.3.2, Decision 7.18). From this point forward, we refer to the execution of action t_α followed by action t_ω as the execution of the (labeled) task t. We denote $\mathcal{T} = \mathcal{T}_\alpha \cup \mathcal{T}_\omega \cup \{\tau\}$, where $\tau \notin \mathcal{T}_\alpha \cup \mathcal{T}_\omega$. ⌟

Choice

Behavior can be conditionally executed, as shown in Figure 7.1(c) where the choice 8 fingers? tests if we have sufficient ladyfingers. The outcome of a choice is based on the evaluation of a data expression. These data expressions may consist of values, variables and functions. The concrete syntax of a choice consists of a split diamond that is closed by a merge diamond. The alternative conditional behavior (branch) is specified in between the split and the merge diamond. Therefore every branch in the choice is syntactically finite. A branch is selected according to the outcome of the evaluating function. Every outcome corresponds to one annotated label of an edge, specified in between squared brackets. The precedence relation for the split diamond is denoted on the incoming edge and holds for all the outgoing edges. The precedence relation for the merge diamond is denoted on the outgoing edge and holds for all the incoming edges.

Decision 7.5: We map a branch i to its corresponding process term p_i, and define an operator across branches. We assume a decision function d that maps every evaluation to exactly one branch in the process term. According to this description we obtain:

$$\bigvee_d \langle p_1, \ldots, p_n \rangle$$

⌟

Decision 7.6: The choice operator acts on the state of a system. This means that we require a mechanism to store the actual state. Let Λ denote the set of all values and let $\mathcal{V}$ denote the set of all variables. Then $\Sigma = \mathcal{V} \to \Lambda$ denotes the set of all variable valuations. A variable valuation is a total function that captures the values of the variables. Now, $\sigma \in \Sigma$ denotes a variable valuation where σ is the state vector that stores the variable valuation observed by the system's behavior. The signature of the evaluated decision function is specified as $d : \Sigma \to \mathbb{N}$, where every valuation corresponds to exactly one alternative process term. ⌟

Concurrent Execution

To maximize the output of a system under control, one ideally likes to execute tasks concurrently. Execution can be forked and joined using multiple incoming/outgoing precedence relations, possibly having different precedence relations. Forking concurrent behavior (i.e., the start of the concurrent execution) is depicted by the two

outgoing transitions of the task boil sweet milk in Figure 7.1(b). Joining concurrent behavior (i.e., the end of the concurrent execution) is depicted by the two incoming transitions for the task whisk milk & egg yolk in Figure 7.1(b). Note that a join may be depicted in such a way that the concurrent behaves originates from two (or more) different forks. Likewise, a fork may be depicted in such a way that the concurrent behavior ends in two different joins.

Decision 7.7: We cannot assume that concurrent behavior is started by one fork, and is ended by another (single) join. Hence we duplicate the labeled tasks and force the synchronization in such a way that it corresponds to the concurrent execution described by the structure of the forks and joins. Let p and q be process terms and let $\coprod$ be the synchronized execution operator. So, we specify the synchronized execution as:

$$p \coprod q$$

Informally, this term states that for the task labels that occur in both the terms p and q, we force their synchronized execution. For labeled tasks that occur in either the term p or the term q we allow the interleaved execution as long as the precedence relations are respected. If we want to specify a forking task then we specify it as the first task in both the term p and q. If we want to specify a joining task then we specify it as the last task in both the term p and q. Tasks that need to be performed concurrently, need to be specified between the fork and join task and may only appear at one side of the operator. The use of this operator and the formal syntax is clarified in Figure 7.2 (Chapter 7.2.4). ⌟

Decision 7.8: This decision extends Decision 7.4 where start and finish actions are implicitly considered to be unique. Since task labels are now duplicated to capture the forking and joining of behavior, it is possible that these labels become indistinguishable. For example, when two tasks carry the same label and appear on both sides of the operator, but should be executed interleaved, they are now synchronized. To resolve this, we refine the definition for a labeled task t by extending it with a unique identifier $i \in \mathbb{N}$ such that t_α becomes t^i_α and t_ω becomes t^i_ω. While diagrams are syntactically finite, the set of unique labels that need to be assigned can also be chosen finite. ⌟

Composition

A subplan combines a set of tasks into a named group, thereby enabling the reuse and nesting of behavior. It is represented by a square labeled box that contains behavior in the form of labeled tasks. In Figure 7.1, Make Tiramisu, Make Cream Topping and Make Layers are subplans. Subplans may refer to other subplans (including itself). A reference is represented by a smaller square labeled box that does not contain tasks.

Decision 7.9: We introduce process equations to facilitate composition. Let $\mathcal{S}$ denote the set of subplan labels, disjoint from the set of task labels, i.e., $\mathcal{S} \cap \mathcal{T} = \emptyset$. Furthermore, we require that in every subplan all of the subplan references are unique. This

assumption guarantees that all tasks can be uniquely identified. Let $A \in \mathcal{S}$ describe the behavior for process term p by the equation

$$A \equiv p$$

In the equation provided above, the process term p denotes the modeled implementation of subplan A. ⌟

Decision 7.10: This decision extends Decision 7.8, because process equations may refer inside a subplan to another subplan, therefore making the tasks potentially indistinguishable. To illustrate the uniquification of tasks, consider a single task label that is used and instantiated in two different subplans. We assume $P : List(\mathcal{S})$ to be a list of subplan labels at which actions t^i_α and t^i_ω are extended such that during execution we observe $t^{i,P}_\alpha$ and $t^{i,P}_\omega$.

If we consider the left-hand side of the equation as the parent node and the element on the right as its child, we infer a tree-like structure on subplans. Every node in the tree stores the label of a subplan reference. If a node has children, we know that every directly attached child is uniquely labeled, because we assume that all subplan references inside a subplan are unique. Now, if we construct a path of from a leave node to the root, we know that this path is unique. Since this path instantiates P, we know that all tasks can be uniquely identified by a path.

Note that P only appears during execution, because it is computed during the execution. So if we write a specification, we omit the writing of P. ⌟

Decision 7.11: To mark the initial process we introduce a special keyword **init** that marks the initial process term p. We assume that every specification has exactly one initialization, which is expressed by

$$\textbf{init}\ p$$

⌟

Resource

A task consumes a set of resource labels when it starts its execution and produces a set of resource labels when it finishes. Both sets can be empty. All tasks with the same label are of the same (proto)type: they produce and consume the exact same amount of each resource label. In the DSL, these definitions are stored separately as shown for task add mascarpone in Table 7.1(d). Note that some resource labels (such as wooden spoon) are *used* by consuming them at the start and returning them at the end of a task.

Decision 7.12: We assume that a task claims all required resources during its entire execution. So, an early resource release is not considered. Let $\mathcal{R}$ denote the finite set of resource labels. Now, the function $R_Q : \mathcal{T}_\alpha \to (\mathcal{R} \to \mathbb{N})$ denotes the (possibly empty) set of resources that are required to start the execution of a task. Similarly,

the function $R_P : \mathcal{T}_\omega \to (\mathcal{R} \to \mathbb{N})$ denotes the (possibly empty) set of resources that are produced when the execution of a task finishes. The amount of resources that are available is denoted by the function $R_A : \mathcal{R} \to \mathbb{N}$. The resource usage is encoded in the state vector as the reserved variable R_A in σ, whereas R_Q and R_P are globally given. ⌟

7.2.3 Derived Formal Syntax, Taxonomy and Static Semantics

Next we provide in Table 7.1 a summary of the derived formal syntax and taxonomy for the DSL. Here, p_{legacy} and $p_{\text{semantics}}$ denote placeholders terms for the additional syntax that are required to describe the legacy language construct (Chapter 7.2.5) and assign the semantics (Chapter 7.3.3), respectively. The descending binding strength of operators is defined as:

$$\text{“}\cdot\text{”}, \text{“}\lll\text{”}, \text{“}\coprod\text{”}, \text{“}\bigvee_d\text{”}$$

Of these operators “$\lll$”, “$\coprod$” and “$\cdot$” associate to the right. Priorities can be overruled by using the parentheses “(” and “)”.

process term	operator name	variable	description
$p ::= \tau$	skip		
$t_\alpha^{i,P}$	start of a task t	$i : \mathbb{N}$ $P : List(\mathcal{S})$	finite identifier list of process definition labels
$t_\omega^{i,P}$	finish of a task t	$i : \mathbb{N}$ $P : List(\mathcal{S})$	finite identifier list of process definition labels
$p \cdot p$	sequential composition		
$p \lll p$	preemptive sequential composition		
$\bigvee_d \langle p_1, \ldots, p_n \rangle$	conditional choice	$d : \Sigma \to \mathbb{N}$ $n : \mathbb{N}$	decision function finite branch number
$p \coprod p$	synchronized parallel composition		
$A \equiv p$	process definition	$A \in \mathcal{S}$	
p_{legacy}	grammar for legacy language constructs		
$p_{\text{semantics}}$	grammar for defining semantics		
init p	the initial executed process term		

Table 7.1 Formal abstract syntax and taxonomy for the DSL

The static semantics of the DSL is described in Table 7.2. Here, we introduce a set of variables $\mathcal{V}$ and a set of values Λ, by which we can construct the set of all possible valuations Σ. The resource management is represented by the state vector $\sigma : \Sigma$. Resources are claimed and released by tasks. Hence we introduce $\mathcal{T}_\alpha$ and $\mathcal{T}_\omega$ that respectively denote the collection of task starts and task finishes. The entire collection of tasks, including the internal action, is denoted by $\mathcal{T}$. To map the available resources, the required resources and the produced resources per task, we specify R_A, R_Q and R_P, respectively. To specify subplan references, we introduce the set of process labels $\mathcal{S}$.

symbol	description
$\mathcal{V}$	set of all variables
Λ	set of all values
$\Sigma = \mathcal{V} \rightarrow \Lambda$	set of all variable valuations
$\sigma : \Sigma$	state vector
$\mathcal{T}_\alpha$	finite set of task starts
$\mathcal{T}_\omega$	finite set of task finishes
$\mathcal{T}$	finite set of tasks, where $\mathcal{T}_\alpha \cup \mathcal{T}_\omega \cup \{\tau\}$ and $\tau \notin \mathcal{T}_\alpha \cup \mathcal{T}_\omega$
$R_A : \mathcal{R} \rightarrow \mathbb{N}$	argument of σ; denoting the available quantity per resources
$R_Q : \mathcal{T}_\alpha \rightarrow (\mathcal{R} \rightarrow \mathbb{N})$	denoting the required resources to start the execution of a task
$R_P : \mathcal{T}_\omega \rightarrow (\mathcal{R} \rightarrow \mathbb{N})$	denoting the produced resources at the finish of a task
$\mathcal{S}$	finite set of process labels where $\mathcal{S} \cap \mathcal{T} = \emptyset$

Table 7.2 Static semantics for the DSL

7.2.4 Validation of the Formal Syntax

At this point, domain experts and language engineers are involved to mature the formal abstract syntax. So we first transform the concrete syntax into the formal abstract syntax, where after we sat down with the involved experts and engineers to validate the derived syntax and discussed the intended semantics.

For illustrative purposes we reconsider the subplans from Figure 7.1 and write them in the formal abstract syntax as shown in Figure 7.2. To obtain the specification all nodes are transformed to start tasks. Since every task label occurs only once, we omit (for presentation purposes) their identifiers. All finish–start precedence relations are specified by the $\cdot$ operator. All start–start precedence relations are specified by the $|||$ operator. The choice operator is replaced by the $\bigvee$ having an arity of two, because the choice depicts two alternatives. Furthermore we specify three equations and an initialization. The equations specify the behaviors for the different subplans. The initialization denotes the subplan that is executed first. The concurrent behavior is specified with the help of the synchronized operator $||$. Here the forking and joining of tasks are duplicated in the way described in Decision 7.7.

To validate the completeness and the soundness of the syntactic notions, we compared the notions with the intended behavior by composing them into different terms and showed the resulting behavior using a prototype implementation of the semantic engineering bridge (Chapter 8). Using this approach, we detected five ambiguities. Since the language is provided under a non-disclosure agreement, we are allowed to only illustrate two of them. The remaining ambiguities concern non-disclosed parts in the language. To show the ambiguities we reuse the Tiramisu example and moderate some part for illustration purposes.

The first ambiguity is illustrated by Figure 7.1(b). Here we moderate a part of the Tiramisu example by replacing the sequential composition between reduce mixture (B.4) and enrich mixture (B.5) with a preemptive sequential composition. We keep the sequential composition between B.5 and Cool down topping (B.6) such that we get B.4 $|||$ (B.5 $\cdot$ B.6). The result is shown in Figure 7.3(a). Next, we define E $\equiv$ B.4 $|||$ B.5 as illustrated in Figure 7.3(b). Based on a syntactic replacement on the concrete syntax level, domain experts expect from both Figure 7.3(a) and Figure 7.3(b) that $\mathsf{B.6}_\alpha$ can occur before $\mathsf{B.4}_\omega$. However in Figure 7.3(b), the DSL's

$$
\begin{aligned}
\text{Make Tiramisu} \equiv\ & ((\text{Make Cream Topping} \ \|\!\|\!\| \ \text{line loaf pan}_\alpha \cdot \text{Make Layers}) \ \amalg \\
& (\text{Make Cream Topping} \cdot \text{add mascarpone}_\alpha \cdot \text{Make Layers}) \ \amalg \\
& (\text{Make Cream Topping} \ \|\!\|\!\| \ \text{mix coffee syrup}_\alpha \cdot \text{Make Layers})) \cdot \\
& \text{cool down cake}_\alpha \cdot \text{present and serve}_\alpha
\end{aligned}
$$

$$
\begin{aligned}
\text{Make Cream Topping} \equiv\ & ((\text{boil sweet milk}_\alpha \ \|\!\|\!\| \ \text{egg yolk mixture}_\alpha \cdot \\
& \text{whisk milk \& egg yolk}_\alpha) \ \amalg \ (\text{boil sweet milk}_\alpha \cdot \\
& \text{whisk milk \& egg yolk}_\alpha)) \cdot \text{reduce mixture}_\alpha \cdot \\
& \text{enrich mixture}_\alpha \cdot \text{cool down topping}_\alpha
\end{aligned}
$$

$$
\text{Make Layers} \equiv \bigvee_{8\ \text{fingers?}} \langle\ \tau,\ \text{dip fingers}_\alpha \cdot \text{cover with cream}_\alpha \cdot \text{Make layers}\ \rangle
$$

init Make Tiramisu

Figure 7.2 Subplans and their initialization in the DSL's formal syntax

(legacy) implementation performs a mathematical substitution such that brackets are placed around E. This changes the dynamic behavior, since E needs to successfully terminate completely before $\mathsf{B.6}_\alpha$ is performed, i.e., the legacy behavior is described by $(\mathsf{B.4} \ \|\!\|\!\| \ \mathsf{B.5}) \cdot \mathsf{B.6}$.

Decision 7.13: Changing the execution semantics of existing operators adversely effects the behavior on the validated and implemented concrete legacy models. To preserve the backward compatibility for concrete models of the informal DSL, a new "all finish-start" precedence relation is added. We use a directed edge annotated with an fs^ label. This precedence relation can only be used in conjunction with a subplan reference. Let p be a reference and q be any process term then the all finish-start behavior is described by $(p) \cdot q$. ⌟

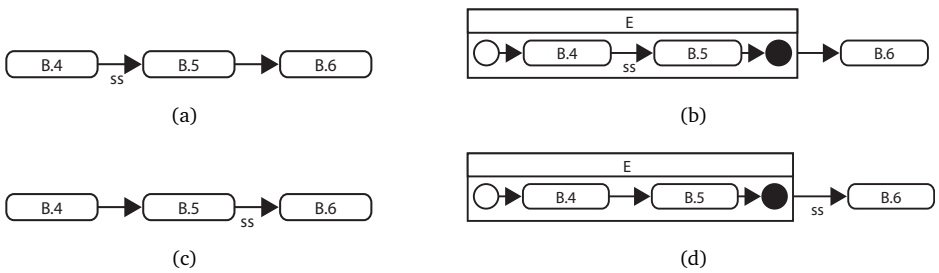


Figure 7.3 Ambiguity on finish–start and start–start relations

The second ambiguity considers a similar fragment, namely $\mathsf{B.4} \cdot \mathsf{B.5} \ \|\!\|\!\| \ \mathsf{B.6}$ which is shown in Figure 7.3(c). Here, domain experts expect that $\mathsf{B.4}_\omega$ is followed by $\mathsf{B.5}_\alpha$,

and $\mathsf{B}.5_\alpha$ is followed by either $\mathsf{B}.5_\omega$ or $\mathsf{B}.6_\alpha$. Now, we define $\mathsf{E} \equiv \mathsf{B}.4 \cdot \mathsf{B}.5$ and write $\mathsf{E} \lll \mathsf{B}.6$ as shown in Figure 7.3(d). By considering a syntactic replacement, we expect that $\mathsf{B}.6_\alpha$ occurs no earlier than after performing $\mathsf{B}.5_\alpha$. Domain engineers however expected that $\mathsf{B}.6_\alpha$ can occur after performing $\mathsf{B}.4_\alpha$. Engineers consider a composite term is 'in progress' as soon as some start action is observed and *not* when all start actions have been observed.

Decision 7.14: To preserve the backward compatibility for concrete models written in the informal DSL, a new "any start-start" precedence relation is added. We use a directed edge annotated with an ss$ label. This precedence relation can only be used in conjunction with subplan references. Let p be a reference and q be any process term then the any start-start behavior is expressed by

$$p \lfloor\!\lfloor q$$

⌟

7.2.5 Formal Syntax for Legacy Constructs

Based on Decision 7.13 and Decision 7.14 we instantiate placeholder term p_{legacy} with its corresponding grammar. Because only Decision 7.13 adds a new operator, the grammar is extend by the $\lfloor\!\lfloor$ operator. The other decision is covered by previously defined constructs, i.e., placing parentheses.

process term	operator name	variable	description
$p_{\text{semantics}} ::= \quad p \lfloor\!\lfloor p$	left merge composition		

Table 7.3 Formal abstract syntax for expressing the DSL's legacy language constructs

Adding the $\lfloor\!\lfloor$ operator affects the binding strength of the operators. Hence, we redefine the precedence rules:

$$\text{“}\cdot\text{”}, \text{“}\lfloor\!\lfloor\text{”}, \text{“}\lll\text{”}, \text{“}\coprod\text{”}, \text{“}\bigvee_d\text{”}$$

Of these operators "$\lfloor\!\lfloor$", "$\lll$", "$\coprod$" and "$\cdot$" associate to the right. Priorities can be overruled by using parentheses "(" and ")".

7.3 Formalizing Dynamic Semantics

We use SOS to assign dynamic operational semantics to the DSL's process terms (abstract notions). SOS associates a labeled transition system to terms, where action transitions describe the discrete event behavior. While SOS has already been explained in Chapter 2.1, we here only explain the semantic notions and assign semantics to the individual DSL's abstract notions.

7.3.1 Semantic Preliminaries

Process

A process is a tuple $\langle p, \sigma \rangle$, where p denotes a process term for an element of an activity diagram, and $\sigma \in \Sigma$ denotes a variable valuation.

Transition

A transition describes a state change between two processes, thereby observing a possible action that is represented by a label.

Decision 7.15: The displayed information is limited to the executed task and its associated resources during the transition. Hence, a label consists of two elements: (i) the label of the executed task and (ii) the associated set of resources. A transition dictates either continuative behavior or successful termination. ⌟

Continuative Action Transitions $: \_ \xrightarrow{-} \_ \subseteq (\mathcal{P} \times \Sigma) \times (\mathcal{X} \times (\mathcal{R} \to \mathbb{N})) \times (\mathcal{P} \times \Sigma)$, where $\mathcal{X}$ is (i) $\mathcal{T}$ when an internal action is performed, or (ii) $\mathcal{T} \times \mathbb{N} \times \mathcal{L}(\mathcal{S})$ when the start of a task is performed. The intuition of an action transition $\langle p, \sigma \rangle \xrightarrow{t,R} \langle p', \sigma' \rangle$ is that the process $\langle p, \sigma \rangle$ performs a discrete action (t, R), thereby transforms into the process $\langle p', \sigma' \rangle$. σ' denotes the corresponding valuation of the process p' after performing the transition t, associating the set of resources R.

Terminating Action Transitions $: \_ \xrightarrow{-} (\checkmark, \_) \subseteq (\mathcal{P} \times \Sigma) \times (\mathcal{X} \times (\mathcal{R} \to \mathbb{N})) \times (\mathcal{P} \times \Sigma)$, where $\mathcal{X}$ is the same as for the continuative action transitions. The intuition of a termination transition $\langle p, \sigma \rangle \xrightarrow{t,R} \langle \checkmark, \sigma' \rangle$ is that the process $\langle p, \sigma \rangle$ transforms into the process $\langle \checkmark, \sigma' \rangle$, by performing the discrete action (t, R). Here $\checkmark$ denotes the successful terminated process.

7.3.2 Operational Semantics

Skip

The internal action τ is defined in Table 7.4 as deduction rule **(skip)**. A τ action is an internal action that cannot be observed nor claim resources.

Decision 7.16: τ does not change the state vector σ, because is does not specify an update to σ. Hence, $\emptyset$ represents the no resource claim $\forall_{r \in \mathcal{R}} \{R(r) = 0\}$, where it uses the auxiliary function R that maps all resource labels to zero. ⌟

Start of a Task

The start of a task is defined in Table 7.4 as deduction rule **(start-task)**. $t_\alpha^{i,P}$ is the action that starts task t. To perform $t_\alpha^{i,P}$, all of the required resources $R_Q(t_\alpha)$ need to be available.

$$\textbf{(skip)}\ \frac{}{\langle \tau, \sigma \rangle \xrightarrow{\tau, \emptyset} \langle \checkmark, \sigma \rangle}$$

$$\textbf{(start-task)}\ \frac{\sigma(R_A) \geq R_Q(t_\alpha)}{\langle t_\alpha^{i,P}, \sigma \rangle \xrightarrow{t_\alpha^{i,P}, R_Q(t_\alpha)} \langle t_\omega^{i,P}, \sigma[R_A \to \sigma(R_A) - R_Q(t_\alpha)] \rangle}$$

$$\textbf{(finish-task)}\ \frac{}{\langle t_\omega^{i,P}, \sigma \rangle \xrightarrow{t_\omega^{i,P}, R_P(t_\omega)} \langle \checkmark, \sigma[R_A \to \sigma(R_A) + R_P(t_\omega)] \rangle}$$

$$\textbf{(FS)}\ \frac{\langle p, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}{\langle p \cdot q, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q \\ p' \cdot q \end{matrix}, \sigma' \right\rangle} \qquad \textbf{(SS1)}\ \frac{\langle p, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark, \sigma' \\ p', \sigma' \end{matrix} \right\rangle}{\langle p \lll q, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q, \sigma' \\ p' \lll q, \sigma' \end{matrix} \right\rangle}$$

$$\textbf{(SS2)}\ \frac{\langle p, \sigma \rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma' \rangle\ ,\quad \langle q, \sigma \rangle \xrightarrow{\beta',\rho'} \left\langle \begin{matrix} \checkmark \\ q' \end{matrix}, \sigma'' \right\rangle,\quad \langle p, \sigma'' \rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma''' \rangle}{\langle p \lll q, \sigma \rangle \xrightarrow{\beta',\rho'} \left\langle \begin{matrix} p \\ p \lll q' \end{matrix}, \sigma'' \right\rangle}$$

$$\textbf{(SS\$)}\ \frac{\langle p, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark, \sigma' \\ p', \sigma' \end{matrix} \right\rangle}{\langle p \lfloor\!\lfloor q, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} q, \sigma' \\ p' \rfloor \emptyset \lfloor q, \sigma' \end{matrix} \right\rangle}$$

$$\textbf{(C)}\ \frac{\langle p_{d(\sigma)}, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}{\langle \bigvee_d \langle p_1, \ldots, p_n \rangle, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}\ d(\sigma) \in [1, n]$$

$$\textbf{(spc)}\ \frac{\langle p \rfloor sync(p) \cap sync(q) \lfloor q, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}{\langle p \lfloor\!\lfloor q, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}$$

$$\textbf{(pe)}\ \frac{\langle p, \sigma \rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle}{\langle A, \sigma \rangle \xrightarrow{\beta[^{P \lhd A}/_P],\rho} \left\langle \begin{matrix} \checkmark \\ p'[^{P \lhd A}/_P] \end{matrix}, \sigma' \right\rangle}\ A = p \in \mathcal{S}$$

Table 7.4 Deduction rules for the DSL's basic operators

Decision 7.17: The resource availability is expressed by the premise $\big(\sigma(R_A) \geq R_Q(t_\alpha)\big)$. As all functions are total, we assume a point-wise evaluation. If $t_\alpha^{i,P}$ is performed, we observe $t_\alpha^{i,P} \in \mathcal{T}_\alpha$ where i is the unique identifier and P is the position in the subplan hierarchy, thereby claiming resources $R_Q(t_\alpha)$. ⌟

Decision 7.18: To ensure that $t_\omega^{i,P}$ follows after $t_\alpha^{i,P}$, we rewrite the term to the finish of the task. The number of claimed resources are subtracted from the available resources. This is reflected by $\sigma[R_A \rightarrow \sigma(R_A) - R_Q(t_\alpha)]$. ⌟

Finish of a Task

The finish of a task is defined in Table 7.4 as deduction rule **(finish-task)**. $t_\omega^{i,P}$ is the action that finishes the task t.

Decision 7.19: Any release of the claimed (produced) resources is added to the set of available resources, reflected by $\sigma[R_A \rightarrow \sigma(R_A) + R_P(t_\omega)]$. The set of premises is empty, so the finish of a task is performed unconditionally. We observe $t_\omega^{i,P}$ and $R_P(t_\omega)$ on the transition and rewrite $t_\omega^{i,P}$ to $\checkmark$ such that the tasks indicate successful termination. ⌟

Sequential Composition

In Table 7.4, deduction rule **(FS)** defines the sequential composition.

Decision 7.20: We follow the standard semantics given in literature, in e.g., [GMR+06]. Here, $p \cdot q$ behaves as q, if p successfully terminates after performing action (β, ρ), i.e., the upper case of in Table 7.4 as deduction rule **(FS)**. If p, by performing action (β, ρ) becomes p', then the process $p \cdot q$ behaves as $p' \cdot q$, i.e., the lower case of in Table 7.4 as deduction rule **(FS)**. ⌟

Preemptive Sequential composition

The preemptive sequential composition is defined in Table 7.4 as deduction rule **(SS1)** and as deduction rule **(SS2)**. Here, we want a right term of the operator to perform actions iff a left term *can* successfully terminate.

Decision 7.21: Deduction rule **(SS1)** defines the behavior when the term p performs a transition. Whenever p successfully terminates, then the process continues as q. If p continues as p', the term continues as $p' \lll q$. Informally, deduction rule **(SS2)** expresses that q can perform an action, iff p *can* terminate but does *not perform* the action yet. $p \lll q$ states that q performs action (β', ρ') such that p stays allowed to successfully terminate by performing action (β, ρ). To ensure continuation of p after the action taken by q in deduction rule **(SS2)**, the premise $\langle p, \sigma'' \rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma''' \rangle$ is added. ⌟

Left Merge Composition

In Table 7.4, deduction rule **(SS$)** defines the left merge composition. The process on the left of the operator has to perform an action first, after which the remaining process behaves concurrently. Note that the concurrency used here is less restrictive, in terms of concurrency, than the $\lfloor\lfloor$ operator.

Decision 7.22: The upper case of deduction rule **(SS$)** expresses that if p successfully terminates in $p \lfloor\lfloor q$ the process behaves as q (no remainder of p can interleave). If p continues as p', the lower case of deduction rule **(SS$)** expresses that the remaining process behaves as $p' \rfloor \emptyset \lfloor q$. To allow reuse, we introduce $p' \rfloor \emptyset \lfloor q$, which takes the tasks that need to synchronize as a parameter. When no tasks need to synchronize the parameter is set to $\emptyset$. A detailed explanation for this auxiliary operator is given in Chapter 7.3.3. ⌟

Conditional Choice

The conditional choice selects a process term according to the outcome of an evaluation function as defined in Table 7.4 as deduction rule **(C)**.

Decision 7.23: Let $d : \Sigma \to \mathbb{N}$ be a surjective function that, provided a state vector σ, returns a value within the domain of the enumeration (which is a subset of $\mathbb{N}$). The outcome of $d(\sigma)$ is forced to be in range by the function. ⌟

Synchronized Parallel Composition

The semantics for the synchronized parallel composition is given in Table 7.4 as deduction rule **(spc)**. If the behavior occurs on both sides of the operator *and* all of the actions are enabled, then the execution is synchronized. If the behavior occurs on only one side, it executes without any synchronization.

Decision 7.24: As terms are rewritten on both sides of the operator, the set of synchronizing actions must be calculated prior to executing any action. The set needs to be preserved until the synchronized parallel composition successfully terminates. For this we use an auxiliary concurrency operator, that is the same operator as the preemptive sequential operator, though instantiated differently. The concurrent execution operator initiates the auxiliary concurrency operator $p \rfloor C \lfloor q$, where it computes $C \subseteq \mathcal{T} \times \mathbb{N} \times List(\mathcal{S})$, being the set of synchronizing actions that occur in both p and q. ⌟

Decision 7.25: To compute C, we introduce function *sync* that computes the intersection of transition labels that both occur in p and q by $sync(p) \cap sync(q)$. We interpret a transition label $\beta \equiv t_x^{i,P}$ as the triple $(t_x, i, P) \in \mathcal{T} \times \mathbb{N} \times List(\mathcal{S})$. The *sync* function

is defined as

$$
\begin{array}{lcl}
sync(\tau) & = & \emptyset \\
sync(t_{\alpha}^{i,P}) & = & \{(t_{\alpha}, i, P)\} \cup sync(t_{\omega}^{i,P}) \\
sync(t_{\omega}^{i,P}) & = & \{(t_{\omega}, i, P)\} \\
sync(p \cdot q) & = & sync(p) \cup sync(q) \\
sync(\bigvee_d \langle p_1, \ldots p_n \rangle) & = & \bigcup_{i=1}^{n} sync(p_i) \\
sync(p \lll q) & = & sync(p) \cup sync(q) \\
sync(p \lll q) & = & sync(p) \cup sync(q) \\
sync(p \coprod q) & = & sync(p) \cup sync(q) \\
sync(A) & = & sync(p') \text{ where } A = p \in \mathcal{S} \text{ and } p' \text{ is obtained by} \\
 & & \qquad \text{substituting all labels } P \text{ by } P \lhd A \text{ in } p
\end{array}
\tag{1}
$$

⌟

Process Definitions

In Table 7.4, deduction rule **(pe)** states the semantics for a process definition. For each task in a process, we generate a unique identifier by taking the list of identifier equations (the scope in which an action is executed) and combine it with the task's identifier. The generation of such an identifier is determined during execution by substituting the hierarchical levels in tasks.

Decision 7.26: The substitution is performed by taking the current hierarchical level P and append the identifier's equation $P \lhd A$. The result is then assigned to the action and the remaining process term. To illustrate, we evaluate term $D = a^i$ at the hierarchy level c, which claims no resources. If we perform the substitution we observe the transition $a^{i,[c \lhd D]}, \emptyset$ during the execution. ⌟

7.3.3 Auxiliary Operational Semantics

This subsection describes the operational semantic for the syntactic notations that have been introduced while assigning the semantics to the abstract syntax. As these notions could not be captured by the already defined semantics, the auxiliary operational semantics extends the semantics for the abstract syntax. The syntax is extended by the syntactic notions, which are displayed in Table 7.5.

process term		operator name	variable	description
$p_{\text{semantics}} ::=$	$p \rfloor C \lfloor p$	concurrent execution	C	set of actions
	$C \lfloor p$	encapsulation operator	C	set of actions

Table 7.5 Definition for the DSL's process term $p_{\text{semantics}}$

Decision 7.27: We extend Decision 7.25 by adding the following two equations to the synchronization function. These functions are used in the operational semantics of the auxiliary operators.

$$\begin{array}{lcl} sync(C \lfloor p) & = & C \cup sync(p) \\ sync(p \rfloor C \lfloor q) & = & C \cup sync(p) \cup sync(q) \end{array} \tag{2}$$

⌟

Concurrent Execution

The concurrent execution $p \rfloor C \lfloor q$ only synchronizes behavior, if an action β occurs in C and both p and q have the action enabled. Otherwise, if β does not occur in C, enabled actions from both p and q are performed without synchronization.

If we consider the action $\beta \notin C$ and p or q having action β enabled, then in Table 7.5 deduction rule **(spe5)** and deduction rule **(spe6)** define that if either p or q successfully terminate in $p \rfloor C \lfloor q$, they respectively continue as $C \lfloor p$ or $C \lfloor q$. In Table 7.5, deduction rule **(spe7)** and deduction rule **(spe8)** define that if either p or q continue as p' or q' in $p \rfloor C \lfloor q$, they respectively continue as $p' \rfloor C \lfloor q$ or $p \rfloor C \lfloor q'$.

If action $\beta \in C$, then in Table 7.5 deduction rule **(spe1)** states that if p and q can perform the action β, and both end up in a terminating state, then $p \rfloor C \lfloor q$ ends up in a terminating state after executing β. In Table 7.5, deduction rule **(spe2)** and deduction rule **(spe3)** state that if either p or q ends up in a terminating state and both processes perform an action β, they continue as a right synchronized execution $C \lfloor p'$ or $C \lfloor q'$, respectively. In Table 7.5, deduction rule **(spe4)** states that if p and q both have action β enabled and continue as p' and q', $p \rfloor C \lfloor q$ continues as $p' \rfloor C \lfloor q'$. In all cases C remains constant.

Decision 7.28: Deduction rules **(spe2)** and **(spe3)** dictate encapsulation ($\lfloor$), which is undefined within the current semantics. Therefore we introduce an auxiliary operator, for which we assign semantics. ⌟

Encapsulation Operator

The encapsulation operator $C \lfloor p$, prohibits the execution of all actions that occur in C. The semantics is provided in Table 7.5 as deduction rule **(encap)**, where the successful termination of $C \lfloor p$ is denoted in the upper case, and the continuation of $C \lfloor p$ as $C \lfloor p'$ in the lower case.

7.3.4 Validation of the Formal Semantics

To validate the assigned semantics, we have used the framework that is presented in Chapter 8. With the help of this framework, we could automatically generate state spaces for the DSL's models, using the above stated syntax, the corresponding semantics and the mCRL2 toolset [GMR+06]. The generated state spaces have been

$$\textbf{(spe1)}\ \frac{\beta \in C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle}$$

$$\textbf{(spe2)}\ \frac{\beta \in C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle p', \sigma'\rangle, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor p', \sigma'\rangle}$$

$$\textbf{(spe3)}\ \frac{\beta \in C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle q', \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor q', \sigma'\rangle}$$

$$\textbf{(spe4)}\ \frac{\beta \in C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle p', \sigma'\rangle, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle q', \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle p' \rfloor C \lfloor q', \sigma'\rangle}$$

$$\textbf{(spe5)}\ \frac{\beta \notin C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor q, \sigma'\rangle} \qquad \textbf{(spe6)}\ \frac{\beta \notin C, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle \checkmark, \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle C \lfloor p, \sigma'\rangle}$$

$$\textbf{(spe7)}\ \frac{\beta \notin C, \langle p, \sigma\rangle \xrightarrow{\beta,\rho} \langle p', \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle p' \rfloor C \lfloor q, \sigma'\rangle} \qquad \textbf{(spe8)}\ \frac{\beta \notin C, \langle q, \sigma\rangle \xrightarrow{\beta,\rho} \langle q', \sigma'\rangle}{\langle p \rfloor C \lfloor q, \sigma\rangle \xrightarrow{\beta,\rho} \langle p \rfloor C \lfloor q', \sigma'\rangle}$$

$$\textbf{(encap)}\ \frac{\langle p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ p' \end{matrix}, \sigma' \right\rangle, \beta \notin C}{\langle C \lfloor p, \sigma\rangle \xrightarrow{\beta,\rho} \left\langle \begin{matrix} \checkmark \\ C \lfloor p' \end{matrix}, \sigma' \right\rangle}$$

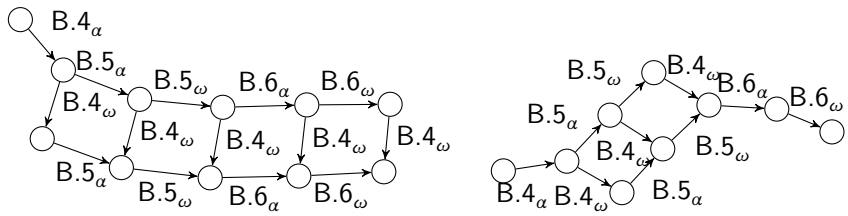
Table 7.6 Deduction rules for the DSL's auxiliary operators

visualized and validated by observing the possible execution scenarios. With the help of the framework, we were also able to point out the differences between the behavior executed by the models depicted in Figure 7.3(a) and Figure 7.3(b). The concrete state spaces for this example are visualized in Figure 7.4(a) and Figure 7.4(b)*. The entire state space for the Tiramisu example can be generated and be visualized in a similar fashion.

7.4 Related Work

The formalized DSL is inspired on the UML [RJB04] modeling format. As such, the formalization of UML Statechart Models [DMY02, JS04], UML State Machines [Kus01, PL99], UML Sequence Diagrams [Are02], and UML Activity Diagrams [BCR00] can

*The superscript notations have been removed for presentation purposes.



(a) Intended behavior (b) Observed behavior

Figure 7.4 Generated LTSs for a discovered disambiguation

be considered as starting points. Here, design constraints are captured by the Object Constraint Language (OCL) [CW02]. As DSLs add domain specific notions, the usability of these existing formal definitions are significantly reduced depending on the complexity and nature of the changes. In our case, the non-disclosed changes are quite extensive and include scheduling, dispatching logic, exception handling and more. When considering TRECS as a separate language, rather than one specialized from UML, we find many frameworks and methods that *transform* DSLs and/or their concrete models in such a way that formal syntax and semantics is assigned [MWW04].

The framework of [EW05] restricts the modeling languages in a way that only descriptions of possible domain configurations are mapped. Firstly, domain (ontological) semantics is assigned to language constructs. Secondly, the ontological assumptions are identified by administering the elements of the domain and their relationships. Thirdly, the ontological assumptions are transferred and become the rules that restrict the use of the language constructs and limit the statements to the specific domain. Finally, they construct the meta-model from these rules.

A similar approach is taken by [JS09] where the meta-model is formalized bottom-up. They start from a simple core that defines the syntax for a class of DSLs. Next, a relating class, i.e., transformation, is defined to relate syntactic elements of one domain with elements of another. Then, a special element is introduced that generates all the domains for a particular class, i.e., the meta-model. Finally, formal Horn logic [Hor51] is added to preserve and formulate various properties over the different domains using the FORMULA [JS08] theorem prover. In the work of [EW05, JS09], a meta-model is created that describes the constructs that specify the commonalities and/or differences between DSLs. Meta-models are expressed using OCL and class diagrams that define relationships [CCR08]. Our route is similar, since we take basic notions and create a syntactical meta-model for them. Rather than constraining class diagrams, we provide an actual model of computation through SOS. This allows us to specify the behavior *mathematically* for each syntax element in isolation and provide a compositional language.

The work of [CCGT09] shows a pragmatic and instrumented approach towards providing operational semantics for DSLs. They sketch how the semantics in an

axiomatic, operational, or denotational (translational) manner, based on the DSL's assigned taxonomy. Based on the selected adequate target language, a mapping is provided that preserves the semantic relation. Our approach is similar but we use operational semantics instead. Operational semantics is preferred when considering the semantics of complex, composed language terms. In [Wol09] operational semantics in the MDSE are explored for a small academic language. Since we demonstrate feasibility of the operational approach for a large, industrial language, other aspects (like backward compatibility) need to be considered. Our work supersedes this scope and complements both approaches.

Work of [AvdBE11] shows yet another approach. The authors prototype the semantics of a DSL, called SLCO [ABE10]. With the help of the ASF+SDF Meta-Environment [BvDH+01], SLCO models are transformed into an LTS. In work of [AvdBEV12], the authors again take the SCLO language and transform it to an LTS. However they now provide the transformation in the MOF [ISO05] and EMF [BBM03] framework. These approaches can be considered as alternative options.

Finally, in [DRJK+06, SW09] different approaches are taken to assign dynamic semantics to DSLs in the context of MDSE. Here, dynamics are assigned through Abstract State Machines (ASMs) [Bör98], with extensions to Prolog [Wie03] and Scheme [IEE91]. As the underlying semantics of ASMs is formally defined through SOS [Ton98], we demonstrate that intermediate semantic definitions may be omitted.

7.5 Conclusions

This chapter illustrates a structured approach to formalize the (dynamic) semantics of an industrial DSL using SOS.

We start from an existing DSL with an informal and an implicit semantics. We first identify the concrete notions for the concrete models. This results in a structuring of the concrete syntax. Furthermore, it facilitates the generalization of concrete syntax elements and syntax variation points. These observations enable multiple concrete syntax projections in the near future.

Once identified, concrete notions are then projected onto abstract notions where the concrete syntax is mimicked as closely as possible. By starting with the most elementary notions, we try to reuse abstract notations where possible. If the reuse is not possible we try to refine existing abstract notions. We introduce new abstract notions when refinement is not possible. This approach helps in creating a compositional language. However, it also results in (many) orthogonal annotations, such as the start for a task, process scopes, and task identifiers. These annotations are required to obtain observable and uniquely distinguishable actions in the formal semantics.

The formal (dynamic) semantics for the abstract notions are captured by SOS deduction rules. Because SOS is a compositional formalism, it facilitates an incremental approach where the behavior of simple notions are composed into a more complex and compound behavior. As the semantics is subjected to numerous design decisions, we had to introduce auxiliary operators to *exactly* capture the semantics. The formal-

ization has led to the formalization of over thirty abstract notions, roughly covering 75% of the DSL.

The semantics has been assigned in consultation with engineers. By displaying the exhibited formal execution and comparing the execution to the engineer's *intended* behavior and the performed execution by the *implemented* interpreter, we identified five semantic gaps for which two of them are discussed and addressed in this chapter. To close the cognitive gap between the intended and the implemented semantics, we needed to introduce additional complementary operators.

In the formal semantics, "available resources" (R_A) could replace the state vector (σ), since all evaluations on the examples are performed for R_A. We decided to explicitly define σ since the full DSL contains other constructs that also manipulate σ and influence the decision taking process. Moreover, we choose to define resource claims using total mappings (visible on the transition label). This implies that all resource labels need to be known in advance. Finally, we want to emphasize that the DSL allows to fork and join concurrency in an almost arbitrary manner. In turn, this implied a significant refinement on the notions to obtain unique task labels and ensure correct synchronization.

After the formalization, the suggested operators to resolve the ambiguities have been manually implemented in the DSL's interpreter. For each use of a legacy operator, domain experts now have to decide to either retain the legacy operator or to switch to the new operator based on the disambiguated semantics. This approach provides backward compatibility with the (execution behavior) of the informal language. Note that the use of complementary operators reduced the regression and qualification impact significantly while phasing out ambiguous behavior.

Chapter 8

Defining a Semantic Bridge

8.1 Introduction

Structural Operational Semantics (SOS) [Plo04] assigns semantics to syntax with the help of deduction rules that describe the allowed set of actions that belong to a particular process. Although the notation is practical for describing the language's behavior, it is unpractical for verification purposes. That is, there are hardly any suitable automated transformation techniques that allow the transformation of SOS specifications (along with a syntactical instance) to models that facilitate forms of verification.

This chapter formulates a systematic approach that closes the gap by transforming the signature of the syntax, the SOS' deduction rules, and a language specific model into a symbolic representation of a labeled transition system, called a *Linear Process Specification* (LPS) [BBG97, Fok07]. The LPS can be subjected to formal analysis, e.g., simulation, explicit labeled transition system generation, and verification. The transformation that is described in this chapter is restricted to deduction rules that are in the De Simone-format [dS85].

The LPS is chosen as the target formalism, because it (i) has a mathematical representation that can capture the SOS' deduction rules and (ii) can be directly implemented in the mCRL2 language [GMR+06, Sofb]. In fact, the LPS serves as a backbone for the representation and manipulation of behavioral models in the mCRL2 toolset. Since this toolset facilitates a higher-order term rewrite system, the execution of behavior and other transformation tools, we are able to exhaustively explore state spaces and conduct profound analyses for these LPSs.

The approach aims to transform any formal behavioral specification into a specification that is suitable for a formal analysis, e.g., simulation and model-checking. The technique can be used to prototype formally defined DSLs, to investigate the behavior dictated by the underlying operational semantics, or enables the transformation of any formal language into a valid mCRL2 specification. This chapter only discusses the

transformation. Reflections on usability and efficiency of the method are separately discussed in Chapter 10

Chapter 8.2 describes the construction of a semantic bridge. Chapter 8.3 states the correspondence relation between the semantic bridge and the operational semantics of the subjected language. Chapter 8.4 demonstrates the approach for a small language. Chapter 8.5 provides a few recommendations when implementing an mCRL2 model. Chapter 8.6 shows how the semantic bridge deals with predicates. Chapter 8.7 describes how rule format extensions can be modeled. Chapter 8.8 highlights some of the work that has been performed by other authors. Chapter 8.9 briefly concludes.

8.2 Method

The method provides a template that transforms a Transition System Specification (TSS) into an LPS. The LPS is described in the mCRL2 notation, which is a symbolic description of the transition relations (transition system) described by a TSS.

To perform the transformation we require a TSS. Explicitly a TSS defines the signature of the terms for which the semantics are assigned. A TSS also incorporates the deduction rules that define the semantics for the syntactical expressions. We assume that any model that is transformed adheres to a TSS' term signature.

The method translates a TSS into an LPS. An LPS consists of several components, for which the sort component encodes the different sorts used in the TSS and the TSS' term signature. The collection of the LPS' data equations are used to compute the set of transition relations that are enabled. The LPE generates the transitions. The (abstract) model that corresponds to the TSS initializes the LPS.

The transformation is restricted to the deduction rules that comply to the De Simone format. This format is chosen since it is one of the elementary formats for describing SOS deduction rules [MRG07]. Chapter 9 demonstrates a transformation for the deduction rules that are provided in a more elaborate format, i.e., the Extended Tyft format [Gal03].

To directly use the LPS in the mCRL2 toolset, we sometimes slightly deviate from notations that are common in mathematics, e.g., when denoting a set comprehension. The framework that we present is restricted to the use of an *mCRL2-restrictive* TSS, for which the restrictions are provided below.

Definition 8.2.1 (mCRL2-restrictive TSS)**.** A TSS is *mCRL2-restrictive* if

1. the signature Σ contains finitely many function symbols,
2. the set of labels $\mathcal{A}$ is finite,
3. the set of deduction rules $\mathcal{D}$ is finite, and
4. the conditions of the deduction rules need to be expressed by mCRL2 data expressions.

Chapter 8.6 discusses several possibilities for relaxing some of these restrictions.

8.2.1 Signature Transformation

For a signature Σ that consists of the different function symbols $f_1, \ldots, f_n$, we define a sort $\mathcal{P}$ together with some additional functions in the mCRL2 language by:

$$\begin{array}{lll} \textbf{sort} & \mathcal{P} = & \textbf{struct}\ f_1(\pi_1 : \mathcal{P}, \ldots, \pi_{ar(f_1)} : \mathcal{P})?is_{f_1} \\ & & \vdots \\ & & \mid f_n(\pi_1 : \mathcal{P}, \ldots, \pi_{ar(f_n)} : \mathcal{P})?is_{f_n}; \end{array}$$

For terms of this sort, $f_1, \ldots, f_n \in \mathcal{C}$ are the constructor functions. The projection functions $\pi_i \in \mathcal{M}$ are used to retrieve argument i of a function symbol. These functions are defined by the equations $\pi_i(f(x_1, \ldots, x_{ar(f)})) = x_i$ in case $i \leq ar(f)$ and undefined otherwise. The recognizer functions $is_{f_i} \in \mathcal{M}$ facilitate the evaluation whether a term is of a particular form. The equations defining recognizer function is_{f_i} are $is_{f_i}(f_i(x_1, \ldots, x_{ar(f_i)})) = true$ and $is_{f_i}(f_j(x_1, \ldots, x_{ar(f_j)})) = false$ for $i \neq j$. For sort $\mathcal{P}$ equality is denoted by $\approx$. Since all sorts in an LPS need to be represented finitely, item 1 from Definition 8.2.1 needs to hold for modeling $\mathcal{P}$.

8.2.2 Transition Relations

The transition relation models pairs that consist of a label and a term. The transition relation is represented by $\mathcal{R}$. We assume that the set of action labels, say $\{a_1, \ldots, a_n\}$, is represented by a sort $\mathcal{A}$.

$$\begin{array}{ll} \textbf{sort} & \mathcal{A} = \textbf{struct}\ a_1 \mid \cdots \mid a_n; \\ \textbf{sort} & \mathcal{R} = \textbf{struct}\ relation(\pi_{\mathrm{l}} : \mathcal{A}, \pi_{\mathrm{t}} : \mathcal{P}); \end{array}$$

The projection functions π_{l} and π_{t} are used to respectively retrieve the transition label and process term for a transition relation. Recognizer functions are not specified, because they are irrelevant for the transformation. Since all sorts in an LPS need to be represented finitely, item 2 from Definition 8.2.1 needs to hold for modeling $\mathcal{A}$.

We introduce a function R that satisfies the property, for all $s, s' \in \mathcal{C}(\Sigma)$ and labels $l \in \mathcal{A}$

$$relation(l, s') \in R(s) \qquad \text{iff} \qquad s \xrightarrow{l} s'$$

Strictly speaking, we transform the mathematical representation of s, s', l on the right of *iff* into the counterpart mCRL2 representation on the left.

Every relation transition must be derived from the conclusion at the level of the initial source term, i.e., at the bottom of the proof tree. Let $\mathcal{D}$ be the set of all deduction rules that are described by the TSS, represented by $\{d_1, \ldots, d_n\}$. Then we introduce for each of deduction rule $d \in \mathcal{D}$ a function $R_d : \mathcal{P} \to Set(\mathcal{R})$ that computes the transition relations for deduction rule d given a process term. All the transition relations that are valid for a process term are computed with the help of function $R : \mathcal{P} \to Set(\mathcal{R})$:

$$\begin{array}{ll} \textbf{map} & R : \mathcal{P} \to Set(\mathcal{R}); \\ \textbf{var} & p : \mathcal{P}; \\ \textbf{eqn} & R(p) = R_{d_1}(p) \cup \cdots \cup R_{d_n}(p); \end{array}$$

Because we compute $R_{d_1}(p) \cup \cdots \cup R_{d_n}(p)$, we require that $\mathcal{D}$ contains a finite (presentable) set of deduction rules. Hence, item 3 from Definition 8.2.1 must hold.

Definition 8.2.1 (De Simone **format).** A TSS $(\Sigma, \mathcal{D})$ is in De Simone format, when every deduction rule $d \in \mathcal{D}$ complies to the following form:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t}[Cond_d]$$

where all of $x_1, \ldots, x_{ar(f)}$ and y_i, for $i \in I$ are distinct variables, $f \in \Sigma$, $I \subseteq \{1, \ldots, ar(f)\}$, and t is a process term that only contains variables from $\{x_j \mid j \notin I\} \cup \{y_i \mid i \in I\}$ and does not have repeated occurrences of variables, l_i's and l are labels and $Cond_d$ is a condition on the labels of the premises and the label of the conclusion.

Now let d be a deduction rule in the De Simone format and for each $d \in \mathcal{D}$, we introduce a data equation R_d. Its informal explanation of the structure and the definition of the introduced auxiliary functions are provided after the data equation. The formal explanations of the auxiliary functions are described in the following paragraphs.

The data equation R_d computes the set of transition relations that holds for deduction rule d:

$$\begin{array}{lll}
\textbf{map} & R_d : \mathcal{P} \to Set(\mathcal{R}); & \\
\textbf{var} & p : \mathcal{P}; & \\
\textbf{eqn} & R_d(p) = & \{ \quad s : \mathcal{R} \\
 & & \mid \quad is_f(p) \\
 & & \wedge \quad \sigma^t(\pi_t(s)) \\
 & & \wedge \quad \exists_{l_1,\ldots,l_{|I|}}(Cond_d(l_1, \ldots, l_{|I|}, \pi_1(s)) \\
 & & \qquad \wedge \bigwedge_{i \in I} y_i \in vars(t) \Rightarrow relation(l_i, \mu^t_{y_i}(\pi_t(s))) \in R(\pi_i(p)) \\
 & & \qquad \wedge \bigwedge_{i \in I} y_i \notin vars(t) \Rightarrow \exists_{z_i : \mathcal{P}} relation(l_i, z_i) \in R(\pi_i(p))) \\
 & & \wedge \quad \bigwedge_{j \notin I} x_j \in vars(t) \Rightarrow \mu^t_{x_j}(\pi_t(s)) \approx \pi_j(p)\};
\end{array}$$

The body consists of several conjuncts. An element is added to the set of transition relations when all conjuncts of the body hold. Here, the conjunct $is_f(p)$ states that the rule can only be applied to terms p that are headed by the function symbol f. The conjunct $\sigma^t(\pi_t(s))$ states that the target term must have the same structure as the term t from the deduction rule (see *Check Target Structure* below). The third, fourth and fifth conjunct state that labels l_i and terms y_i need to be found such that the condition and premises of the deduction rule are satisfied (see *Capture Conditions* below). The third conjunct states that we require a transition relation that fulfills the condition. The fourth and the fifth conjunct restrict the possible transition relations to those that agree with the substitution for the occurrences of x_i and y_i in t to obtain $\pi_t(s)$ (see *Extract Variable Instance* below). The expression $\mu^t_x(p)$ denotes the term (from p) that instantiates the variable x in t. The last condition checks that the substitutions of the source variables, occurring in the target, are those provided by p.

Check Target Structure

The transition relation's target term requires to be an instance of the term t. So, we define a function $\sigma^t : \mathcal{P} \to \mathbb{B}$ that asserts this requirement. If t is of the form x for some variable p then we introduce the following equation:

var $p : \mathcal{P}$;
eqn $\sigma^x(p) = true$;

and for t of the form $f(t_1, \ldots, t_{ar(f)})$ for some function symbol f and terms $t_1, \ldots, t_{ar(f)}$ we introduce the equation:

var $p : \mathcal{P}$;
eqn $\sigma^{f(t_1,\ldots,t_{ar(f)})}(p) = is_f(p) \wedge \sigma^{t_1}(\pi_1(p)) \wedge \cdots \wedge \sigma^{t_{ar(f)}}(\pi_{ar(f)}(p))$;

Here the auxiliary functions $\sigma^{t_i} : \mathcal{P} \to \mathbb{B}$ for $1 \leq i \leq ar(f)$ correspond to the target structure checks for the terms t_i.

Capture Conditions

The user who performs the transformation needs to introduce the functions $Cond_d$ that capture the meaning of the conditions in the deduction rules. Hence, the applicability is restricted to the conditions that can be expressed by mCRL2 data equations. The functions take a set of action labels and rewrite them to a Boolean expression. The computability of these expressions are bound to the solvability of the underlying rewriter. All data expressions must be stated in the mCRL2 syntax.

map $Cond_d : \mathcal{A} \times \cdots \times \mathcal{A} \times \mathcal{A} \to \mathbb{B}$;

For practical cases, these functions can be easily captured by the mCRL2 data language and are computable by the mCRL2 rewriter. When a condition function is introduced item 4 from Definition 8.2.1 must hold.

Extract Variable Instance

To retrieve the terms that instantiate a variable x in the term t, we introduce the projection function $\mu_x^t : \mathcal{P} \to \mathcal{P}$. When t is of the form x we introduce the following data equation:

map $\mu_x^x : \mathcal{P} \to \mathcal{P}$;
var $p : \mathcal{P}$;
eqn $\mu_x^x(p) = p$

When t is of the form $f(t_1, \ldots, t_{ar(f)})$ for some function symbol f and terms $t_1, \ldots, t_{ar(f)}$, we introduce for each $1 \leq i \leq ar(f)$ a data equation such that x occurs in t_i:

map $\mu_x^{f(t_1,\ldots,t_{ar(f)})} : \mathcal{P} \to \mathcal{P}$;
var $p : \mathcal{P}$;
eqn $\mu_x^{f(t_1,\ldots,t_{ar(f)})}(p) = \mu_x^{t_i}(\pi_i(p))$

Additionally, we add the auxiliary functions $\mu_x^{t_i} : \mathcal{P} \to \mathcal{P}$ with their corresponding equations. Note that we only use μ_x^t in those cases where $x \in vars(t)$. Hence it is irrelevant that the function μ_x^t is not defined for variables different from x that do not occur in t. Because we only consider t in which every variable occurs at most once, μ_x^t is well-defined. To illustrate the extraction of variable instances consider Example 8.1.

Example 8.1(Extract Variable Instance Data Equations). Let $t = \oplus(x, y, z)$ be a resulting term to which the conclusion of a deduction rule rewrites, where $\oplus$ denotes a function symbol and x, y, z are instantiated variables. To extract the values from the instantiated variables we define the following data equations:

map $\mu_x^x, \mu_y^y, \mu_z^z, \mu_x^{\oplus(x,y,z)}, \mu_y^{\oplus(x,y,z)}, \mu_z^{\oplus(x,y,z)} : \mathcal{P} \to \mathcal{P};$
var $p : \mathcal{P};$
eqn $\mu_x^x(p) = p;$
$\mu_y^y(p) = p;$
$\mu_z^z(p) = p;$
$\mu_x^{\oplus(x,y,z)}(p) = \mu_x^x(\pi_1(p));$
$\mu_y^{\oplus(x,y,z)}(p) = \mu_y^y(\pi_2(p));$
$\mu_z^{\oplus(x,y,z)}(p) = \mu_z^z(\pi_3(p));$

Here $\pi_1, \ldots, \pi_3$ denote the projection functions that respectively retrieve the first, second and third argument of $\oplus(x, y, z)$. $\triangle$

8.2.3 Linear Process Transition Generator

The generation of transitions is captured by the specification's LPE. Basically, transitions are performed as long as the set of transition relations belonging to term p is non-empty. So we declare process X with the process parameter $p : \mathcal{P}$. Then, for every iteration we select a transition relation r such that $r \in R(p)$ holds. Subsequently, for each r we dispatch the transition, i.e., $\pi_l(r)$, and update term p in the next state to be $\pi_t(r)$. So we specify the following LPE:

proc $X(p : \mathcal{P}) = \sum_{r:\mathcal{R}} r \in R(p) \to \pi_l(r) \cdot X(\pi_t(r));$

The behavior associated with a term p is specified through process $X(p)$:

init $X(p);$

8.3 Correspondence

The following theorem expresses the correspondence between a labeled transition system associated with a closed process term and an mCRL2 process $X(p)$. The proof of this theorem is stated in Appendix A.2.

Theorem A.2.5 (Correspondence)**.** Let $(\Sigma, \mathcal{D})$ be an mCRL2-restrictive TSS in the De Simone format. Then for every $p \in \mathcal{C}(\Sigma)$, the labeled transition system associated with p and the labeled transition system associated with $X(p)$ are isomorphic.

8.4 Application

To illustrate the details and the applicability of the approach, we consider the process algebra MPT from [BBR09], extended with the parallel composition operator. The example assumes that the set of actions is finite, i.e., $\mathcal{A} = \{a_1, \ldots, a_n\}$. The signature of the language consists of the nullary function symbol 0, the unary function symbols $\alpha.\_$ (for $\alpha \in \mathcal{A}$ and _ denoting the argument), and the binary function symbols $\_ + \_$ and $\_ \parallel \_$. For representing the binary function symbols in the deduction rules, this section uses the infix notation (instead of a prefix notation). By applying the signature transformation we get:

$$
\begin{array}{ll}
\textbf{sort} & \mathcal{P} = \textbf{struct}\ \ zero?is_{zero} \mid a_0(\pi_1 : \mathcal{P})?is_{a_1} \mid \cdots \mid a_n(\pi_1 : \mathcal{P})?is_{a_n} \\
 & \qquad \mid alt(\pi_1 : \mathcal{P}, \pi_2 : \mathcal{P})?is_{alt} \mid par(\pi_1 : \mathcal{P}, \pi_2 : \mathcal{P})?is_{par}; \\
\textbf{sort} & \mathcal{A} = \textbf{struct}\ a_1 \mid \cdots \mid a_n; \\
\textbf{sort} & \mathcal{R} = \textbf{struct}\ relation(\pi_1 : \mathcal{A}, \pi_t : \mathcal{P});
\end{array}
$$

where *zero*, $a_i(\_)$, $alt(\_,\_)$ and $par(\_,\_)$ represent 0, $a_i.\_$, $\_ + \_$, and $\_ \parallel \_$ respectively. The deduction rules for the MPT process algebra are:

$$
(a_1)\frac{}{a_1.x_1 \xrightarrow{a_1} x_1} \qquad \cdots \qquad (a_n)\frac{}{a_n.x_1 \xrightarrow{a_n} x_1} \qquad \textbf{(a1)}\frac{x_1 \xrightarrow{l} y_1}{x_1 + x_2 \xrightarrow{l} y_1}
$$

$$
\textbf{(a2)}\frac{x_2 \xrightarrow{l} y_2}{x_1 + x_2 \xrightarrow{l} y_2} \qquad \textbf{(p1)}\frac{x_1 \xrightarrow{l} y_1}{x_1 \parallel x_2 \xrightarrow{l} y_1 \parallel x_2} \qquad \textbf{(p2)}\frac{x_2 \xrightarrow{l} y_2}{x_1 \parallel x_2 \xrightarrow{l} x_1 \parallel y_2}
$$

As no conditions other than *true* appear in the deduction rules we do not consider them in the remainder of this section. To accommodate the (auxiliary) computation we introduce the following functions and variables:

$$
\begin{array}{ll}
\textbf{map} & R, R_{a_1}, \ldots, R_{a_n}, R_{\mathbf{a1}}, R_{\mathbf{a2}}, R_{\mathbf{p1}}, R_{\mathbf{p2}} : \mathcal{P} \to Set(\mathcal{R}); \\
 & \sigma^{x_1}, \sigma^{x_2}, \sigma^{y_1}, \sigma^{y_2}, \sigma^{y_1 \| x_2}, \sigma^{x_1 \| y_2} : \mathcal{P} \to \mathbb{B}; \\
 & \mu^{x_1}_{x_1}, \mu^{x_2}_{x_2}, \mu^{y_1}_{y_1}, \mu^{y_2}_{y_2}, \mu^{y_1 \| x_2}_{y_1}, \mu^{y_1 \| x_2}_{x_2}, \mu^{x_1 \| y_2}_{x_1}, \mu^{x_1 \| y_2}_{y_2} : \mathcal{P} \to \mathcal{P}; \\
\textbf{var} & p : \mathcal{P};
\end{array}
$$

The sort $\mathcal{R}$ refers to the declaration defined in Chapter 8.2.2. The overall transition relation function is defined as:

$$
\textbf{eqn} \quad R(p) = R_{a_1}(p) \cup \cdots \cup R_{a_n}(p) \cup R_{\mathbf{a1}}(p) \cup R_{\mathbf{a2}}(p) \cup R_{\mathbf{p1}}(p) \cup R_{\mathbf{p2}}(p);
$$

To illustrate the relationship between the deduction rules and the data equations we consider the deduction rules for the action prefix **(a1)** and **(p1)**. For presentation

purposes we only state data equations within the simplified set comprehensions. The resulting data equations for an action prefix terms ($\alpha \in \mathcal{A}$) are:

$$
\begin{array}{ll}
\textbf{eqn} & \sigma^{x_1}(p) = \mathit{true}; \\
& \mu^{x_1}_{x_1}(p) = p; \\
& R_\alpha(p) \;\; = \{r : \mathcal{R} \mid \mathit{is}_\alpha(p) \wedge \sigma^{x_1}(\pi_{\mathrm{t}}(r)) \wedge \mu^{x_1}_{x_1}(\pi_{\mathrm{t}}(r)) \approx \pi_1(p)\};
\end{array}
$$

The required equations for the deduction rule **(a1)** are:

$$
\begin{array}{ll}
\textbf{eqn} & \sigma^{y_1}(p) = \mathit{true}; \\
& \mu^{y_1}_{y_1}(p) = p; \\
& R_{\mathbf{a1}}(p) = \;\; \{r : \mathcal{R} \mid \mathit{is}_{\mathit{alt}}(p) \wedge \sigma^{y_1}(\pi_{\mathrm{t}}(r)) \\
& \qquad\qquad \wedge \exists_{l:\mathcal{A}}(\mathit{relation}\left(l, \mu^{y_1}_{y_1}(\pi_{\mathrm{t}}(r))\right) \in R(\pi_1(p)) \wedge \;\; \pi_{\mathrm{l}}(r) \approx l)\};
\end{array}
$$

To model the deduction rule **(p1)** the following set of equations (including the auxiliary ones) is constructed:

$$
\begin{array}{ll}
\textbf{eqn} & \sigma^{y_1 \| x_2}(p) = \mathit{is}_{\mathit{par}}(p) \wedge \sigma^{y_1}(\pi_1(p)) \wedge \sigma^{x_2}(\pi_2(p)); \\
& \sigma^{y_1}(p) \quad\;\; = \mathit{true}; \\
& \sigma^{x_2}(p) \quad\;\; = \mathit{true}; \\
& \mu^{y_1 \| x_2}_{y_1}(p) = \mu^{y_1}_{y_1}(\pi_1(p)); \\
& \mu^{y_1}_{y_1}(p) \quad\;\; = p; \\
& \mu^{y_1 \| x_2}_{x_2}(p) = \mu^{x_2}_{x_2}(\pi_2(p)); \\
& \mu^{x_2}_{x_2}(p) \quad\;\; = p; \\
& R_{\mathbf{p1}}(p) \quad = \;\; \{r : \mathcal{R} \mid \mathit{is}_{\mathit{par}}(p) \wedge \;\; \sigma^{y_1 \| x_2}(\pi_{\mathrm{t}}(r)) \\
& \qquad\qquad \wedge \exists_{l:\mathcal{A}}(\mathit{relation}\left(l, \mu^{y_1 \| x_2}_{y_1}(\pi_{\mathrm{t}}(r))\right) \in R(\pi_1(p)) \\
& \qquad\qquad \wedge \mu^{y_1 \| x_2}_{x_2}(\pi_{\mathrm{t}}(r)) \approx \pi_2(p)) \wedge \;\; \pi_{\mathrm{l}}(r) \approx l\};
\end{array}
$$

The deduction rules **(a2)** and **(p2)** are analogous modeled to the deduction rules **(a1)** and **(p1)**.

To perform a meaningful analysis for the closed term p, we provide the following LPE:

$$
\textbf{proc} \quad X(p : \mathcal{P}) = \sum_{r:\mathcal{R}} r \in R(p) \rightarrow \pi_{\mathrm{l}}(r) \cdot X(\pi_{\mathrm{t}}(r));
$$

The LPS specification is instantiated by p:

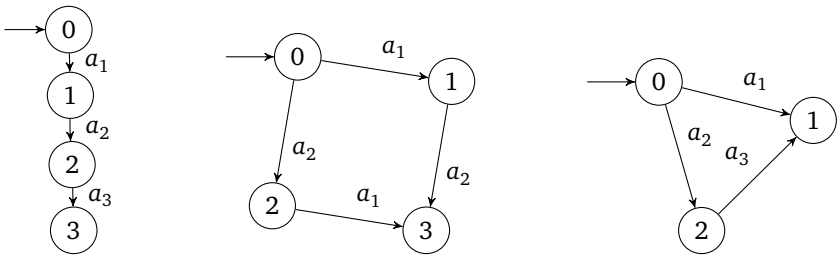
$$
\textbf{init} \quad X(p);
$$

To illustrate that the method is effective, Figure 8.1 provides some graphs that are generated by the mCRL2 toolset (Release 2012, February) for which the models have been obtained using the aforementioned approach. The captions state the initial process terms that has been used to generate the LTSs. The tools that subsequently have been used to generate the pictures are:

1. txt2lps: This tool reads a textual LPS and stores it into the binary LPS format.

2. lps2lts: This tool exhaustively explores an LPS and stores the result in an LTS.

3. ltsview: This tool visualizes the LTS using a spring layout algorithm. The tool has been used to visualize and export the LTS.



(a) $a_1(a_2(a_3(zero)))$ (b) $par(a_1(zero), a_2(zero))$ (c) $alt(a_1(zero), a_2(a_3(zero)))$

Figure 8.1 Three generated LTSs for different MPT SOS input models

8.5 Implementation

The implementation requires a finite number of deduction rules and a finite signature, such that we can generate a finite textual specification. Furthermore we have to apply two restrictions to conduct an analysis. The first restriction applies to the use of actions. The second restriction applies to the use of quantifiers.

In the example we use elements of sort $\mathcal{A}$ (part of the data specification) as actions in an mCRL2 specification. Within the mCRL2 language the direct use of data sorts as action labels is prohibited. To overcome the limitation, we declare a (dummy) action with a data parameter of sort $\mathcal{A}$ and use the data parameter to encode the concluding transition relation from the TSS. This means that instead of $p \xrightarrow{a} p'$, we get $p \xrightarrow{Trans(a)} p'$, where *Trans* is the dummy action label and we use $a \in \mathcal{A}$ as its data parameter*.

The second restriction applies to the use of quantifiers. The mCRL2 language allows existential quantifiers ($\exists$) for which it can solve quantifiers that are reduced into Skolem normal form using the Skolemization method. If quantifiers cannot be reduced into a normal form they are enumerated. Adversely, quantifiers that cannot be reduced into the Skolem normal form and range over an infinite domain may have infinite many solutions, which cannot be property exhaustively simulated.

Recall the data equation that models a deduction rule in Chapter 8.2.2. Here we used the existential quantifiers z_i if we were only interested in the transition and

*For presentation purposes we have removed the dummy actions from the examples in the previous sections.

not the corresponding term. Note, that these quantifiers are not per se necessarily. Therefore, the expressions $\exists_{z_i} relation(l_i, z_i) \in R(\pi_i(p))$ can be simplified to the expressions $l_i \in R^l(\pi_i(p))$, where the function R^l is like R but instead of returning a set of transition relations (consisting of labels and terms) it returns only a set of labels. Let $R^l : \mathcal{P} \to Set(\mathcal{A})$ be the derived function that uses the auxiliary functions $R^l_d : \mathcal{P} \to Set(\mathcal{A})$ for the deduction rules $d \in \mathcal{D}$. Then we define $R^l = \bigcup_{d \in \mathcal{D}} R^l_d$ and specify the auxiliary functions as:

eqn $$R^l_d(p) = \{a : \mathcal{A} \mid is_f(p) \wedge \exists_{l_1,\ldots,l_{|I|}} (Cond_d(l_1, \ldots, l_{|I|}, a) \wedge \bigwedge_{i \in I} (l_i \in R^l(\pi_i(p))))\};$$

8.6 Predicate Extension

A predicate is an expression of a semantic expression. [GV92] shows that predicates are coded as binary relations, i.e., a predicate is a statement that is either true or false. Predicates serve various purposes and have different representations, e.g., divergence [AH89], enabledness [BPW93], probabilistic behavior [LS92], priorities [CH90], etc. They are used to express behavioral properties, like termination and divergence and useful addition to TSSs [BV93].

Because predicates can be encoded into the De Simone-format, we here explain how it can be accomplished using multiple transition relations. Basically every predicate introduces a (different) transition relation function R. With the help of different dummy actions in the LPE, we can observe whether predicates.

Predicate Modeling

To illustrate how predicates are used in deduction rules consider the following two deduction rules. We assume that $\mathbb{P}$ is a predicate and $\longrightarrow$ is a transition relation. When d is a deduction rule with function symbol f, the rules are represented by:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{\mathbb{P}x_j \mid j \in J\}}{\mathbb{P}f(x_1, \ldots, x_{ar(f)})} [Cond_d]$$

or

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{\mathbb{P}x_j \mid j \in J\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t} [Cond_d]$$

where all of $x_1, \ldots, x_{ar(f)}$ and y_i, for $i \in I$ are distinct variables, $f \in \Sigma$, $I, J \subseteq \{1, \ldots, ar(f)\}$ and $I \cap J = \emptyset$, t is a process term that only contains variables from $\{x_k \mid k \notin I \cup J\} \cup \{y_i \mid i \in I\}$ that does not have repeated occurrences of variables, and l_i's and l are labels and $Cond_d$ is a condition on the labels of the premises and the conclusion (if any).

A predicate can be considered as a special kind of transition relation with a special transition label. Therefore, we introduce a special transition relation symbol $\xrightarrow{\mathbb{P}}$ for

predicate $\mathbb{P}$. Then the above deduction rules are represented by:

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{x_j \xrightarrow{\mathbb{P}} y_j \mid j \in J\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{\mathbb{P}} f(z_i, \ldots, z_{ar(f)})} [Cond_d]$$

$$\text{where } z_i = \begin{cases} x_i & \text{if} \quad i \notin J \wedge i \notin I \\ y_i & \text{if} \quad i \in J \wedge i \notin I \end{cases}$$

or

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\} \cup \{x_j \xrightarrow{\mathbb{P}} y_j \mid j \in J\}}{f(x_1, \ldots, x_{ar(f)}) \xrightarrow{l} t} [Cond_d]$$

Because I and J are disjoint the rules remain in the De Simone-format.

The two transition relations are modeled by two transition relations functions. The first transition relation is defined through the function R, such that for all $s, s' \in \mathcal{C}(\Sigma)$ and labels $l \in \mathcal{A}$, holds:

$$relation(l, s') \in R(s) \qquad \text{iff} \qquad s \xrightarrow{l} s'$$

The second transition relation, that models the predicate relation, is defined through function R_{Pred}, such that for all $s, s' \in \mathcal{C}(\Sigma)$ and labels $\mathbb{P} \in \mathcal{A}_{Pred}$, holds:

$$relation(\mathbb{P}, s') \in R_{Pred}(s) \qquad \text{iff} \qquad s \xrightarrow{\mathbb{P}} s'$$

When we assume that s and s' are syntactical identical, the predicate relation appears as a self-loop transition in the generated LTS. To emphasize that the action transitions are different from predicate transitions, we use the action label *Trans* to model action transitions and use the action label *Pred* to model predicate transitions.

act $\quad Trans\colon \mathcal{A};$
$\quad\quad\quad Pred\colon \mathcal{A}_{Pred};$

proc $\quad X(p\colon \mathcal{P}) = \sum_{r\colon \mathcal{R}} r \in R(p) \rightarrow Trans(\pi_1(r)) \cdot X(\pi_t(r))$
$\quad\quad\quad\quad + \sum_{r\colon \mathcal{R}} r \in R_{Pred}(p) \rightarrow Pred(\pi_1(r)) \cdot X(\pi_t(r));$

Predicate Application

This example extends the MPT example (Chapter 8.4) with the termination predicate. We also extend the signature with the function symbol 1. Here 1 denotes the successful termination of a process term. The termination predicate is modeled using the $\downarrow$. In the MPT extension it is common to write $x \downarrow$. The deduction rules are defined by:

$$\textbf{(t1)}\frac{}{1 \downarrow} \qquad \textbf{(t2)}\frac{x_1 \downarrow}{x_1 + x_2 \downarrow} \qquad \textbf{(t3)}\frac{x_2 \downarrow}{x_1 + x_2 \downarrow} \qquad \textbf{(t4)}\frac{x_1 \downarrow \quad x_2 \downarrow}{x_1 \parallel x_2 \downarrow}$$

Because predicates are special transition relations, we replace the predicates by the predicate transition relation. So, for the rules above we obtain the following deduction rules:

$$\textbf{(t1)}\ \frac{}{1 \xrightarrow{\downarrow} 1} \qquad \textbf{(t2)}\ \frac{x_1 \xrightarrow{\downarrow} y_1}{x_1 + x_2 \xrightarrow{\downarrow} y_1 + x_2}$$

$$\textbf{(t3)}\ \frac{x_2 \xrightarrow{\downarrow} y_2}{x_1 + x_2 \xrightarrow{\downarrow} x_1 + y_2} \qquad \textbf{(t4)}\ \frac{x_1 \xrightarrow{\downarrow} y_1 \quad x_2 \xrightarrow{\downarrow} y_2}{x_1 \parallel x_2 \xrightarrow{\downarrow} y_1 \parallel y_2}$$

To model the termination predicate, we first extend the signature by adding a nullary constructor function *one* representing the constant 1 and a recognizer function is_{one}:

$$\begin{array}{ll} \textbf{sort} & \mathcal{P} = \textbf{struct}\ \ zero?is_{zero} \mid one?is_{one} \mid a_0(\pi_1 : \mathcal{P})?is_{a_1} \mid \cdots \mid a_n(\pi_1 : \mathcal{P})?is_{a_n} \\ & \qquad \mid alt(\pi_1 : \mathcal{P}, \pi_2 : \mathcal{P})?is_{alt} \mid par(\pi_1 : \mathcal{P}, \pi_2 : \mathcal{P})?is_{par}; \end{array}$$

To model the predicate we add the singleton set of action labels $\mathcal{A}_{Pred} = \{\downarrow\}$ for which we assume $\mathcal{A}_{Pred} \cap \mathcal{A} = \emptyset$. For computing the terminate predicate relation we introduce the function R_{Pred}, that is defined through the four auxiliary functions $R_{\mathbf{t1}}, R_{\mathbf{t2}}, R_{\mathbf{t3}}, R_{\mathbf{t4}}$. The valid relations for the predicates are computed by:

$$\begin{array}{ll}
\textbf{map} & R_{Pred}, R_{\mathbf{t1}}, R_{\mathbf{t2}}, R_{\mathbf{t3}}, R_{\mathbf{t4}} : \mathcal{P} \to Set(\mathcal{R}); \\
\textbf{var} & p : \mathcal{P}; \\
\textbf{eqn} & R_{Pred}(p) = R_{\mathbf{t1}} \cup R_{\mathbf{t2}} \cup R_{\mathbf{t3}} \cup R_{\mathbf{t4}}; \\
& R_{\mathbf{t1}}(p) \ \ = \{r : \mathcal{R} \mid is_{one}(p) \wedge \pi_{\mathrm{t}}(r) \approx p \wedge \pi_{\mathrm{l}}(r) \approx \downarrow\}; \\
& R_{\mathbf{t2}}(p) \ \ = \{r : \mathcal{R} \mid is_{alt}(p) \wedge \sigma^{y_1+x_2}(\pi_{\mathrm{t}}(r)) \\
& \qquad\qquad \wedge\ relation(\downarrow, \mu^{y_1+x_2}_{y_1}(\pi_{\mathrm{t}}(r))) \in R_{Pred}(\pi_1(p)) \\
& \qquad\qquad \wedge\ \mu^{y_1+x_2}_{x_2}(\pi_{\mathrm{t}}(r)) \approx \pi_2(p)\}; \\
& R_{\mathbf{t3}}(p) \ \ = \{r : \mathcal{R} \mid is_{alt}(p) \wedge \sigma^{x_1+y_2}(\pi_{\mathrm{t}}(r)) \\
& \qquad\qquad \wedge\ relation(\downarrow, \mu^{x_1+y_2}_{y_2}(\pi_{\mathrm{t}}(r))) \in R_{Pred}(\pi_2(p)) \\
& \qquad\qquad \wedge\ \mu^{x_1+y_2}_{x_1}(\pi_{\mathrm{t}}(r)) \approx \pi_1(p)\}; \\
& R_{\mathbf{t4}}(p) \ \ = \{r : \mathcal{R} \mid is_{par}(p) \wedge \sigma^{y_1 \| y_2}(\pi_{\mathrm{t}}(r)) \\
& \qquad\qquad \wedge\ relation(\downarrow, \mu^{y_1 \| y_2}_{y_1}(\pi_{\mathrm{t}}(r))) \in R_{Pred}(\pi_1(p)) \\
& \qquad\qquad \wedge\ relation(\downarrow, \mu^{y_1 \| y_2}_{y_2}(\pi_{\mathrm{t}}(r))) \in R_{Pred}(\pi_2(p))\};
\end{array}$$

To illustrate the use of predicates for the approach, consider Figure 8.2. It shows a generated example with the mCRL2 toolset. The initial specification p is shown in the caption. Here the process performs either an action a_1 and a deadlock or performs an action a_2 and terminates successfully. The tools that have used are identical to those in our previous example.

The actual models that have been used to generate the graphs, including the ones of the previous example, are found in Appendix B.4.

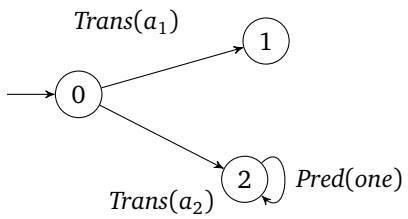


Figure 8.2 Generated LTS for the input model $alt(a_1(zero), a_2(one))$

Remark 8.6.1. Chapter 8.5 discusses the use of a dummy actions to model a data expression as a transition. Here, the dummy actions express the difference between transition relations and predicates relations.

8.7 Rule Format Extensions

This section briefly sketches the SOS rule format extensions that can be incorporated to model different behavior. We consider the use of multi-sorted term signatures, environments, negative premises, and look-ahead transitions.

Multi-sorted signatures are modeled in a similar way as single-sorted signatures, as we have seen in e.g., Chapter 8.2.1. For single sorted signatures all function symbols and variables are of the same sort. Hence, the modeled constructor functions and their arguments that represent the term are of the same sort. In multi-sorted signatures, function symbols and arguments consist of different sorts. By modeling the (appropriate) different sorts, we can deal with multi-sorted signatures. Chapter 9 shows how such a signature is modeled.

Deduction rules sometimes allow that behavior is influenced by data that originates from some environment. Commonly, such an environment is represented by a valuation. If we assume that such an environment E is always present, it can be modeled as a separate process parameter of the LPE. The (transition) relations is then modeled by:

$$relation(l, s', E') \in R(s, E) \qquad \text{iff} \qquad (s, E) \xrightarrow{l} (s', E')$$

This extension modifies the structured sort *relation* and the function R such that they require an environment as an additional argument. Furthermore, they require a sort that models the environment and the functions that provide operations on it. In Chapter 9 we see an example of an environment being a valuation.

Negative premises appeared first in [BB88] to specify the semantics of a priority operator. To model negative premises of the form $s \overset{l}{\not\rightarrow}$, where $l \in \mathcal{A}$, we specify the

(transition) relation as:

$$\forall_{s' \in S} relation(l, s') \notin R(s) \qquad \text{iff} \qquad s \overset{l}{\not\rightarrow}$$

Note, that TSSs with negative premises must be well defined, i.e., stratifiable [Gro93], when the LPS is subjected to an (exhaustive) simulation. This requirement is a language engineer's responsibility and is not detected by the transformation.

Behavior that is affected by any future behavior, it is often modeled through $n, (n \geq 1)$ look-ahead transitions in the premise. If we assume state s^i, $1 \leq i \leq n$ is the input state for the ith-look-ahead transition[†], then we compute the ith transition relation $relation(l^i, s^{i+1})$ by $R(s^i)$. Hence, the chain of n look-ahead transitions is modeled as:

$$relation(l^0, s^1) \in R(s^0) \wedge \ldots \wedge relation(l^n, s^{n+1}) \in R(s^n) \quad \text{iff} \quad s^0 \xrightarrow{l^0} s^1, \ldots, s^n \xrightarrow{l^n} s^{n+1}$$

For transitions and states that do not appear in the conclusion of the deduction rule, we need to add existential quantifiers to find witnesses that are (possibly) used by other look-ahead transitions. The addition of these quantifiers may increase the computational complexity.

8.8 Related Work

SOS meta-theory research is mainly aimed at proving useful properties about TSSs [AFV01, MRG07] such as congruence results [GV92], deriving equational theories [ABV94], conservative extensions [FV98], and soundness of axioms [AIMR09]. Research on how to implement them is underexposed. Most of the related work is performed with the Maude model checker [Sofa]. Other authors have studied the link between the rewriting logic [MOM96] and SOS both from a theoretical [Mes92, Bra01, DGP02, BM05, MRG07] as well as practical point of view [BHMM00, BHMM02, DGP02, VMO02, MR06, VMO06].

In [BHMM02], the outline of a translation from Modular SOS (MSOS) [Mos04a, Mos04b] to the Maude rewriting logic is given and proven correct. The translation is straightforward and the technical twist is in the decomposition of labels, i.e., to the structure of the labels in MSOS. A more elaborate explanation is found in [Bra01]. The work of [VMO06] tries to capture the semantics of Calculus of Communicating Systems (CCS) using rewrites. As these rewrites are labelless, the labels are encoded in the result of a rewrite rule, e.g., the CCS transition of $p \xrightarrow{a} q$ is written as $a.p \longrightarrow \{a\}p$. Though this is a correct transition, $(a.p) \parallel q \longrightarrow (\{a\}p) \parallel q$ is not, since the right-hand side term is not well formed. To overcome this problem, they introduce a dummy operator by which they extend the semantics to generate the transitive closure ([VMO06], pages: 34-38). Basically, rewrites are performed on the outermost function symbol and the result needs to be constructed as such. Since we use tuples

[†]The 0th look-ahead transition is the transition that corresponds to the transition relation from the input state.

to store a relation, rather than encoding it into a single term, we do not suffer from this drawback.

In the works of Mousavi and Reniers [MR06], Verdejo [Ver02], and Verdejo and Marti-Oliet [VMO02, VMO06] we see that the most noticeable difference is the formalism in which they express the TSS. The authors stick to a representation for which hardly any tooling for formal analysis is available, or needs to be developed from scratch. This hinders the possibility to conduct a formal analysis, e.g., model checking. We have chosen the LPS as the target formalism, because it is supported by a collection of tools that are specially aimed at performing formal analysis.

LETOS [Har99] is a tool environment that generates LaTeX documents and executable animations in Miranda [Tur85]. This can be accomplished for a wide range of semantics, including some deterministic SOS forms. Since LETOS only deals with deterministic semantics, it poses some problems when analyzing the behavior of concurrent (non-deterministic) systems.

An approach for implementing SOS rules is presented in [But94], which combines (unconditional) term-rewriting and λ-calculus for simulation. It demonstrates how SOS can be used in proof tools that are based on term rewriting. For that the Larch Prover [GH93] is used, and explained in [But92]. Their method aims at demonstrating and proving the equivalence between different semantics definitions. We, however, aim at creating a bridge that closes the gap between a language for specification and a language for performing analysis. Furthermore, we include conditions, predicates and other rule format extensions, whereas they only allow predicates.

Process Algebra Compiler (PAC) [CMS95] is a tool that takes the signature and the SOS rules of a language and generates a LEX/YACC scanner/parser as well as verification libraries for Lisp and Standard ML. These are then respectively compiled with the kernels of the MAUTO tool [BRdSV89] and the Concurrency Workbench [CS96]. In fact, PAC is a compiler front-end for verification tools. With the help of so-called back-end procedures, it generates the required routines for the different target systems, by relating concepts from the original language to those in the target formalism. The relationship that connects them still needs to be addressed by the user. As our work describes such a relation, this method could be implemented in PAC.

8.9 Conclusions

This chapter demonstrates the transformation for a subclass of SOSs deduction rules, adhering to the De Simone rule format, into a Linear Process Specification in a formal and processable manner. These models can be subsequently analyzed by the mCRL2 toolset. It also expresses how predicates can be modeled, and describes how several rule format extensions can be added.

The semantic bridge still depends (fully) on the formal interpretation and implementation of an engineer. Since we have proven the correspondence relation between a Transition System Specification and a Linear Process, we are confident that

the bridging problem, opposed to an syntactic engineered transformation, results in a transformation that is less prone to interpretation and implementation errors.

Although we have selected mCRL2 as our specification language, we do not foresee any difficulties when choosing another language as long as it has the same expressive power, i.e., the supporting tools facilitate a rewrite system that can compute set comprehensions and provide a transition generator to (exhaustively) explore behavior.

Chapter 9

Applying the Semantic Bridge

9.1 Introduction

This chapter describes a feasibility study that takes a formal specification language, for which the semantics is defined by a TSS and applies the semantic bridge. A defined set of deduction rules is transformed into data equations of an LPS' data specification. The LPE dispatches the transitions for the different transition relations. An instance of the model serves as the initial value for the LPE. Subsequently, the LPS can be subjected to different kinds of analysis, i.e., simulation, state space exploration and verification of modal properties. The idea of the transformation has been discussed in the previous chapter.

There are many formal languages available. So, which one should we select? We could select the SCPL language from Chapter 4. Since we have omitted the semantics, we need to assign the semantics first, like we did in Chapter 7. To avoid the repetition of the formalization process, it is better to select a language that is already formalized. The DSL from Chapter 7 could be a candidate, however parts of the language are proprietary, so we cannot disclose the full language. A better candidate would be Chi 2.0. Since the transformation to the mCRL2 language is discussed in Chapter 5, we could validate that the transformation indeed expresses the prescribed Chi 2.0 semantics. Nevertheless, if we want to know that the denotational approach is correct, we need to assert that the implementation of the mCRL2 toolset implements the mathematical counterparts, i.e., the semantics described by the deduction rules. While the entire thesis is based on the mCRL2 specification language and many case studies have been performed using the mCRL2 toolset, we select the mCRL2 language as our subject of study. We model the deduction rules, restricted to the untimed subset of the mCRL2 language, inside an mCRL2 specification. Hence, we dogfood the mCRL2 toolset its own language [Har06].

To perform the approach, we require (i) a sort that captures the signature of an

mCRL2 process term, (ii) a transformation of the SOS deduction rules into mCRL2 data equations, and (iii) an LPE that performs the (different) transition relations. The domain in which we describe the steps (i), (ii) and (iii) is indicated by the term *meta notation*. The approach from Chapter 8, only discusses the transformation of the deduction rules in the De Simone format [dS85]. Because the mCRL2 language is described in a richer semantics, namely Extended Tyft format [Gal03] the approach is extended by modeling multi-sorted (open) process terms, and format rule extensions that include a data valuation, data parameters in action transitions, multi-actions, functions on action transitions and freshly generated variables.

The outline of this chapter is as following. The mCRL2's language specific design decisions that are incorporated into the semantic approach are described in Chapter 9.2. Chapter 9.3 describes how the deduction rules are modeled. Chapter 9.4 demonstrates some of the models that have been used to validate the correspondence relation between the implementation and the defined semantics of the mCRL2 language. Chapter 9.5 reveals the discovered mismatches in the validation process. Chapter 9.6 describes the limitations that are imposed by the implementation of the semantics. Chapter 9.7 discusses related work. Finally, in Chapter 9.8 we conclude.

9.2 mCRL2 Specific Design Decisions

Even though the mCRL2 language is formally defined in Chapter 2.2, we have to take design decisions such that the semantics can modeled. These decisions are based on the syntax and the semantics of the mCRL2 language. These are provided for the following mCRL2 concepts:

- the deduction rules (Chapter 9.2.1)
- the interpretation of the successful termination $(p, \sigma) \xrightarrow{\alpha} \surd$ (Chapter 9.2.2),
- the modeling of the signature of a process term (Chapter 9.2.3),
- the modeling of data (Chapter 9.2.4),
- the representation of data expressions in the meta notation (Chapter 9.2.5),
- the computation of syntactic multi-actions into semantic multi-action equivalence classes (Chapter 9.2.6),
- the transition relation representation (Chapter 9.2.7).

The transformation describes the concepts as closely as possible. Hence, models are not targeted towards the validation or the verification in the most efficient way. For concepts that cannot be (exhaustively) simulated by their directly modeled counterparts, but have an equivalent notions that can be simulated, we choose the equivalent notions. Concepts that cannot be (exhaustively) simulated are omitted from the transformation. Alternative notions are explicitly stated. The transformation contains

notions that are either model specific or language specific. Language specific notions are modeled *equally* for every mCRL2 model, e.g., operators. Model specific notions may be modeled *differently* between any two mCRL2 models, e.g., the declaration of actions. Hence, Appendix B.5 is dissected into three parts, namely the language specific notions, the model specific notions and the different models that have been used to validate the semantics.

9.2.1 Deduction Rules

SOS deduction rules may describe arbitrary behavior. To ensure that all behavior is modeled by suitable meta notations and poses no threats to the (exhaustive) simulation, the mCRL2's deduction rules are analyzed first.

The TSS of the mCRL2 language is described by a multi-sorted transition relation. It describes (i) a timed action transition labeled with α from state s to s' at time t via $s \xrightarrow{\alpha}_t s'$, (ii) a timed action transition labeled with α from state s to $\checkmark$ at time t via $s \xrightarrow{\alpha}_t \checkmark$, and (iii) the progression of time for state s via $s \rightsquigarrow_t$. As the time domain is dense, i.e., $\mathbb{R}$, it has an uncountable number of values between any two different time values. Thus for any action that is performed in a time interval, or performs a delay $\rightsquigarrow_t$, it results in an uncountable number of time transitions. Without any proper abstraction techniques, it renders any meaningful (exhaustive) simulations impossible. Hence, we restrict the deduction rules to the untimed fragment before we transform the semantics. The untimed fragment corresponds to the black colored deduction rules in Table 2.1, 2.2, 2.4, 2.5, 2.6 and 2.7 in Chapter 2.

Deduction rule Def_2 from Table 2.7 introduces fresh variables w.r.t. σ. The deduction rule assumes an infinite set of variables and imposes no restrictions on the generated fresh variables. Therefore there are infinitely many ways to instantiate $\overrightarrow{v'}$. Thus the recursion operator dictates infinite branching. In theory this kind of behavior poses no problem. In practice, when no abstractions or restrictions are applied, it results in behavior that cannot be (exhaustively) simulated. Based on the exhibited behavior, deduced from tools that exhaustively simulate mCRL2 specifications, we observe that only one $\overrightarrow{v'}$ is generated, for which all of the variables are disjoint from $dom(\sigma)$. For convenience, and the purpose of abstracting from the details of generating fresh variables, we assume given a predicate $fresh : fresh(\sigma, \overrightarrow{v})$ that holds only for those variables $\overrightarrow{v}$ generated by the fresh variable generator. Reflecting this discussion, we redefine the deduction rule for Def_2:

$$(Def_2)\frac{(q[\overrightarrow{v} \mapsto \overrightarrow{v'}], \sigma[\overrightarrow{v'} \mapsto \{[\overrightarrow{d}]\}^{\sigma}]) \xrightarrow{m} (q', \sigma') \qquad fresh(\sigma, \overrightarrow{v'})}{(X(\overrightarrow{v} = \overrightarrow{d}), \sigma) \xrightarrow{m} (q', \sigma')}$$

where $X(\overrightarrow{v} : \overrightarrow{D}) = q \in PE$.

The deduction rules Par_8 and $Sync_4$ silently assume that the values of the non freshly generated variables remain the same. To make the assumption explicitly we introduce

the notation $\sigma' =_{dom(\sigma)} \sigma''$:

$$\sigma' =_{dom(\sigma)} \sigma'' \equiv \forall_{v \in dom(\sigma)} \sigma'(v) = \sigma''(v)$$

Since we have changed the deduction rule Def_2 to generate specific fresh variables, it potentially results to situations where variables from the valuation in the target's premises overlap or variables inside process terms are identical that should have been different. Example 9.1 illustrates the problems for deduction rule (Par_8).

Example 9.1(Fresh Variable Generation). Assume the mCRL2 process $P(v{:}\mathbb{B}) = a_1 \cdot a_2(v)$, and $\mathbf{T} = true$ and $\mathbf{F} = false$ If we model $P(v = \mathbf{F}) \parallel P(v = \mathbf{T})$ and assume σ initially empty, we obtain the following proof tree*:

$$(Par_8)\dfrac{(Def_2)\dfrac{(Seq_2)\dfrac{(ma)\dfrac{}{(a_1, \{v' \mapsto \mathbf{F}\}) \xrightarrow{a_1} \checkmark}}{(a_1 \cdot a_2(v'), \{v' \mapsto \mathbf{F}\}) \xrightarrow{a_1} (a_2(v'), \{v' \mapsto \mathbf{F}\})}}{(P(v = \mathbf{F}), \{\}) \xrightarrow{a_1} (a_2(v'), \{v' \mapsto \mathbf{F}\})} \quad (Def_2)\dfrac{(Seq_2)\dfrac{(ma)\dfrac{}{(a_1, \{v' \mapsto \mathbf{T}\}) \xrightarrow{a_1} \checkmark}}{(a_1 \cdot a_2(v'), \{v' \mapsto \mathbf{T}\}) \xrightarrow{a_1} (a_2(v'), \{v' \mapsto \mathbf{T}\})}}{(P(v = \mathbf{T}), \{\}) \xrightarrow{a_1} (a_2(v'), \{v' \mapsto \mathbf{T}\})}}{(P(v = \mathbf{F}) \parallel P(v = \mathbf{T}), \{\}) \xrightarrow{a_1|a_1} (a_2(v') \parallel a_2(v'), \ldots)}$$

The proof tree clearly illustrates two problems. Firstly, we observe that the conclusion has two freshly chosen variables with the same label. Secondly, we observe that the valuation from the left branch of the tree concludes $\{v' \mapsto \mathbf{F}\}$, whereas the valuation from the right branch of the tree concludes $\{v' \mapsto \mathbf{T}\}$. $\triangle$

To prevent potential variable clashes, all freshly generated variables are renamed from the right premise that overlap with variables of the left premise. To resolve clashes we reuse the mechanism for generating fresh variables.

Let $\overrightarrow{v_{dup}}$ be a list of variables, then we define deduction rule Par_8 as:

$$(Par_8)\dfrac{(p, \sigma) \xrightarrow{m} (p', \sigma'),\ (q, \sigma) \xrightarrow{n} (q', \sigma'') \qquad \sigma' =_{dom(\sigma)} \sigma'' \qquad fresh(\sigma', \overrightarrow{v'}) \qquad fresh(\sigma'', \overrightarrow{v'})}{(p \| q, \sigma) \xrightarrow{m|n} (p' \| (q'[\overrightarrow{v_{dup}} \mapsto \overrightarrow{v'}]), \sigma''')}$$

such that for σ''' holds:

$$\begin{aligned} &\forall_{v \in dom(\sigma')} \sigma'''(v) = \sigma'(v) \wedge \forall_{v \in dom(\sigma'') \setminus dom(\sigma')} \sigma'''(v) = \sigma''(v) \\ &\wedge |\overrightarrow{v_{dup}}| = |\overrightarrow{v'}| \qquad\qquad \wedge \forall_{1 \leq n \leq |\overrightarrow{v_{dup}}|} \sigma'''((\overrightarrow{v'})_n) = \sigma''((\overrightarrow{v_{dup}})_n) \\ &\wedge dom(\sigma''') = dom(\sigma') \cup dom(\sigma'') \cup \{\overrightarrow{v_{dup}}\} \end{aligned}$$

where $\{\overrightarrow{v_{dup}}\}$ is the set interpretation of $\overrightarrow{v_{dup}}$ for which holds $\{\overrightarrow{v_{dup}}\} = (dom(\sigma'') \cap dom(\sigma')) \setminus dom(\sigma)$, and $(\overrightarrow{v})_i$ denotes the ith element of $\overrightarrow{v}$.

Deduction rule $Sync_4$ has a structure similar to Par_8. Hence, we similarly redefine:

$$(Sync_4)\dfrac{(p, \sigma) \xrightarrow{m} (p', \sigma'),\ (q, \sigma) \xrightarrow{n} (q', \sigma'') \qquad \sigma' =_{dom(\sigma)} \sigma'' \qquad fresh(\sigma', \overrightarrow{v'}) \qquad fresh(\sigma'', \overrightarrow{v'})}{(p | q, \sigma) \xrightarrow{m|n} (p' \| (q'[\overrightarrow{v_{dup}} \mapsto \overrightarrow{v'}]), \sigma''')}$$

*Predicate $fresh(\{\}, v)$ is omitted for presentation purposes.

for which the same conditions apply to σ'''.

9.2.2 Successful Termination

Successful termination is denoted by $(p, \sigma) \xrightarrow{\alpha} \checkmark$. It is a contraction of a transition relation and a predicate. Because the transformation is modeled by hand, and successful termination can be modeled in different ways, we here provide four alternatives from which we select one that results in a concise model. Every option consists of two cases. The first transition relation describes the ordinary action transition. The second transition relation describes successful termination.

1. The first option states the one presented in the mCRL2 language:

$$(p, \sigma) \xrightarrow{\alpha} (p', \sigma')$$
$$(p, \sigma) \xrightarrow{\alpha} \checkmark$$

2. The second option presents the predicate by a separate transition relation:

$$(p, \sigma) \xrightarrow{\alpha} (p', \sigma')$$
$$(p, \sigma) \overset{\alpha}{\Longrightarrow} (p, \sigma)$$

3. The third option extends the action label:

$$(p, \sigma) \xrightarrow{\alpha} (p', \sigma')$$
$$(p, \sigma) \xrightarrow{\alpha, \checkmark_l} (p, \sigma)$$

4. The fourth option models the $\checkmark$ predicate as a special process term:

$$(p, \sigma) \xrightarrow{\alpha} (p', \sigma')$$
$$(p, \sigma) \xrightarrow{\alpha} (\checkmark_{\mathbf{p}}, \sigma)$$

In [SRW11b] the authors show how option (1) is modeled. Basically, if a process term successfully terminates, we compute a transition relation (with an irrelevant update state) and one for the termination predicate. If the termination predicate contains a result, we deal with a terminating transition. The action termination function and the transition relation function are nearly identical. This means that all of the deduction rules are modeled twice, i.e., once to compute the transition relation and once to compute the termination predicate.

Option (2) requires an additional transition relation, i.e., the termination relation. Since the solutions for each of the relations are computed separately, we see a similar amount of replication as in option (1).

Option (3) extends the action label with a special label $\checkmark_l$. The extension significantly alters the semantics, i.e., a special label is added to the semantic multi-action. This implies that all deduction rules need to be redefined. Because, we want to model (and study) the current semantics of the mCRL2 language, we do not consider this as a feasible approach.

Option (4) models the predicate as a (special) process term. Based on the transition, we can determine whether a process ends in a successfully terminated state. Since the data valuation is irrelevant in a successfully terminated state, we assume that the data valuation remains unchanged w.r.t. the valuation prior to the transition. This assumption propagates for all deduction rules that describe a successful termination. This omits the modeling of a separate transition relation. Based on these observations we consider that option (4) provides the best solution that can be modeled by hand. Hence, we model the $\checkmark$ predicate as a (special) process term for which we introduce the $\checkmark_{\mathbf{p}}$ process term. By modeling $\checkmark$ as $\checkmark_{\mathbf{p}}$ the language becomes less restrictive. Therefore we assume that $\checkmark_{\mathbf{p}}$ is a special term that cannot be modeled by a user. Moreover it changes the deduction rules' signature slightly, i.e., for non terminating transitions we need to state that the resulting term is not $\checkmark_{\mathbf{p}}$.

Example 9.2(Sequential Composition). This example illustrate the slight alternation in the deduction rules' signature for the rules that belong to the sequential composition. The original rules state:

$$(Seq_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark}{(p \cdot q,\sigma) \xrightarrow{m} (q,\sigma)} \quad (Seq_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma')}{(p \cdot q,\sigma) \xrightarrow{m} (p' \cdot q,\sigma')}$$

By incorporating option (4) the shape of the rules is changed to:

$$(Seq_1)\frac{(p,\sigma) \xrightarrow{m} \checkmark_{\mathbf{p}}}{(p \cdot q,\sigma) \xrightarrow{m} (q,\sigma)} \quad (Seq_2)\frac{(p,\sigma) \xrightarrow{m} (p',\sigma') \quad p' \neq \checkmark_{\mathbf{p}}}{(p \cdot q,\sigma) \xrightarrow{m} (p' \cdot q,\sigma')}$$

Observe that the non-terminating transition explicitly states that the resulting process term p' is unequal to $\checkmark_{\mathbf{p}}$. Note, that other transitions are modeled similarly. △

9.2.3 Process Term

The signature of a process term is modeled by the structured sort $\mathcal{P}$. Every BNF symbol in Definition 2.2.9 introduces a constructor function that carries the textual characterization of the symbol.

Process terms in the mCRL2 language are multi-sorted. The multi-sorted terms are introduced by the arguments of the BNF elements. They describe e.g., the condition in the conditional choice operator, or the action labels that need to synchronize in the communication. For each of these arguments appropriate sorts are introduced. The designated sorts (e.g., $Act_{\Xi}, \mathcal{C}, \mathcal{E}, \mathcal{Q}, \ldots$) are discussed in Chapter 9.3. For now we assume that we know the appropriate sorts and model them accordingly. To model

the $\surd$ predicate, we include the (special) aforementioned process term, modeled via the $\surd_{\mathbf{p}}$ constructor function.

Projection functions are added to access the operands of a term. Here, π_n denotes the nth operand of a process term being of sort $\mathcal{P}$. To access other operands of other sort elements, like the c in a conditional choice, we add projection functions with specially selected names, e.g., π_c. Recognizer functions are added to recognize process terms, which are provided after the question mark (?). These functions only evaluate to *true* iff the term matches the corresponding constructor function.

Based on these decisions we specify the structured sort $\mathcal{P}$ in the meta notation by:

$$
\begin{array}{ll}
\textbf{sort } \mathcal{P} = \textbf{struct} & \\
\quad \surd_{\mathbf{p}}?is_{\surd} & \mid Deadlock?is_{\delta} \\
\quad \mid Alpha(\pi_{multiaction}{:}List(Act_{\Xi}))?is_{\alpha} & \mid Alt(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Alt} \\
\quad \mid Seq(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Seq} & \mid Cond_1(\pi_c{:}\mathcal{E}, \pi_1{:}\mathcal{P})?is_{Cond_1} \\
\quad \mid Cond_2(\pi_c{:}\mathcal{E}, \pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Cond_2} & \mid Sum(\pi_v{:}\mathcal{V}, \pi_1{:}\mathcal{P})?is_{Sum} \\
\quad \mid Par(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Par} & \mid Lmerge(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Lmerge} \\
\quad \mid Sync(\pi_1{:}\mathcal{P}, \pi_2{:}\mathcal{P})?is_{Sync} & \mid Allow(\pi_V{:}Set(Bag(Act_{Lab})), \pi_1{:}\mathcal{P})?is_{Allow} \\
\quad \mid Block(\pi_B{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{Block} & \mid Rename(\pi_{Ren}{:}Act_{Lab} \rightarrow Act_{Lab}, \pi_1{:}\mathcal{P})?is_{Rename} \\
\quad \mid Hide(\pi_I{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{Hide} & \mid Prehide(\pi_U{:}Set(Act_{Lab}), \pi_1{:}\mathcal{P})?is_{Prehide} \\
\quad \mid Comm(\pi_{C^{List}_{comm}}{:}List(\mathcal{C}), \pi_1{:}\mathcal{P})?is_{Comm} & \mid Def(\pi_{PE_{lab}}{:}\mathcal{X}, \pi_{ProcParAsss}{:}List(\mathcal{Q}))?is_{Def};
\end{array}
$$

Discussion

For convenience and presentation purposes, we model a multi-action as a list of actions, i.e., $Alpha(\pi_{multiaction}{:}List(Act_{\Xi}))$. To model the signature precisely requires a separate sort to model a syntactic action, i.e., the sort $\mathcal{M}$ that incorporates the structure of '|' in a syntactic multi-action:

$$
\begin{array}{ll}
\textbf{sort} & \mathcal{M} = \textbf{struct } tau \mid Act(\pi_{a_{lab}}{:}Act_{Lab}, \pi_{args}{:}List(\mathcal{E}) \\
& \qquad \mid Bar(\pi^1_{multiaction}{:}\mathcal{M}, \pi^2_{multiaction}{:}\mathcal{M})?is_{Bar};
\end{array}
$$

So, we should model $Alpha(\pi_{multiaction}{:}\mathcal{M}))$. However, by transforming the '|' into a list of actions we provide a less verbose, but still recognizable structure that represents a syntactic multi-action[†].

9.2.4 Data

Definition 2.2.14 (Chapter 2.2.2) states the semantic interpretation for values, variables, data expressions, lambda expressions, quantifiers and where-clauses. We restrict the interpretation to values, variables and data expressions. Although it is possible to provide suitable implementations for the other concepts, we consider them are out of scope. Moreover, the semantic interpretation of sorts, and their corresponding elements have the same representations as their syntactic counterparts.

[†]For similar reasons semantic multi-action equivalence classes are modeled by lists.

Values

The meta notation presents all values by a single sort Λ[‡]. With the help of constructor functions we represent the different sorts. The sorts that are modeled in the meta notation are derived from the specified sort names in an mCRL2 specification. Let $\mathcal{S}^{\text{mCRL2}}$ be the occurring sort names in an mCRL2 specification, then we model the sort Λ as:

$$\textbf{sort} \quad \Lambda = \textbf{struct}\ Sort^1_\Lambda(s_1{:}Sort_1)?is_{Sort_1} \mid \ldots \mid Sort^n_\Lambda(s_n{:}Sort_n)?is_{Sort_n} \mid \bot;$$

where $Sort_i \in \mathcal{S}^{\text{mCRL2}}$ are the sort labels, $Sort^i_\Lambda$ denotes the constructor function for $Sort_i$, is_{Sort_i} denotes the recognizer function and $\bot$ denotes an undefined value. The order of the constructor functions are analogue to the order in which the sorts are declared in the original mCRL2 specification. This results in a type system where values with their sorts are encoded in a prefix notation, e.g., "$s{:}S$" becomes "$S(s)$". The sorts $Sort_1, \ldots, Sort_n$ are copied from the mCRL2 sort declaration. Hence, we can use the built-in sorts and the user defined sorts in the meta notation. Example 9.3 illustrates the declaration of values (for built-in sorts).

Example 9.3(Sorts in the meta notation). When an mCRL2 specification defines the sorts $\mathbb{B}$ and $\mathbb{N}$, then the structured sort Λ is specified as:

$$\textbf{sort} \quad \Lambda = \textbf{struct}\ \mathbb{B}_\Lambda(b{:}\mathbb{B})?is_\mathbb{B} \mid \mathbb{N}_\Lambda(n{:}\mathbb{N})?is_\mathbb{N} \mid \bot;$$

For the projection functions we have selected appropriate names. To model the Boolean value *true*, we simply write $\mathbb{B}_\Lambda(true)$. △

Variables

Variables, are like values, represented by a single sort that represents all possible sorts. The representation requires two sorts. The first sort $\mathcal{V}_{Lab}$ models the different variable labels. The second sort $\mathcal{V}$ models a variable, where the constructor function indicates its sort, i.e., $\mathcal{V}$ models $\mathcal{X}^{\text{mCRL2}}$. The argument of the constructor function models the designated variable, which retains the option to model the different typed variables in the meta notation. The sorts $\mathcal{V}_{Lab}$ and $\mathcal{V}$ are then modeled as:

$$\textbf{sort} \quad \mathcal{V}_{Lab} = \textbf{struct}\ v_1 \mid \ldots \mid v_n \mid v_\gamma(\pi_{id} : \mathbb{N})?is_{v_\gamma};$$
$$\textbf{sort} \quad \mathcal{V} = \textbf{struct}\ Sort^1_\mathcal{V}(v_L{:}\mathcal{V}_{Lab})?is_{Sort_1} \mid \ldots \mid Sort^n_\mathcal{V}(v_L{:}\mathcal{V}_{Lab})?is_{Sort_n};$$

Every element from $\mathcal{V}_{Lab}$ and $\mathcal{V}$ is derived from an occurring variable in the mCRL2 specification. Hence, we assume that the meta notation contains for every variable $v_i, (1 \leq i \leq n)$ that occurs in the mCRL2 specification a variable v_i. The variables $v_\gamma(m) \notin \{v_i | 1 \leq i \leq n\}$, $(m > 0)$ denote the reserved variables for the generation of fresh variables. The constructor v_γ acts as a prefix for the mth freshly generated variable. Example 9.4 illustrates the declaration of variables in the meta notation.

[‡]Alternatively, values can be modeled by separate sorts.

Example 9.4(Variables in the meta notation). When an mCRL2 specification defines the sorts $\mathbb{B}$ and $\mathbb{N}$, and the variables $b{:}\mathbb{B}$ and $n{:}\mathbb{N}$, then the structured sorts $\mathcal{V}$ and $\mathcal{V}_{Lab}$ are specified as:

sort $\mathcal{V}_{Lab} = \textbf{struct}\ b \mid n \mid v_\gamma(\pi_{id} : \mathbb{N})?is_{v_\gamma}$;
sort $\mathcal{V} \quad = \textbf{struct}\ \mathbb{B}_\mathcal{V}(v_L{:}\mathcal{V}_{Lab}) \mid \mathbb{N}_\mathcal{V}(v_L{:}\mathcal{V}_{Lab})$;

where $v_\gamma(\pi_{id} : \mathbb{N})?is_{v_\gamma}$ corresponds to the labels for the fresh variables. To model the value v of sort $\mathbb{N}$, we simply write $\mathbb{N}_\mathcal{V}(v)$. △

Data Valuation

The meta notation characterizes the data valuation σ as a list of tuples $\overrightarrow{\mathcal{V} \times \Lambda}$. The left element represents a variable and the right element represents a value. In the mCRL2 language it is not possible to model tuples without a constructor. Therefore, we define the structured sort *Argument*, for which *argument* acts as the constructor function for a tuple. The valuation sort is modeled by the sort $\mathcal{S}$:

sort $\mathcal{S} = List(Argument)$;
sort $Argument = \textbf{struct}\ argument(\pi_\mathcal{V}{:}\mathcal{V}, \pi_\Lambda{:}\Lambda)$;

We assume that the elements in data valuation are (increasingly) ordered.

Discussion

A data valuation can be modeled in different ways. So we provide a rationale for our chosen solution. Firstly and preferably, we like to specify the data valuation as $\mathcal{V} \rightarrow \Lambda$. To generate fresh variables for the duplicate variables in e.g., deduction rules Par_4 and Par_8 require enumerations. Since the mCRL2 toolset (currently) provides no support for the enumeration of functions, we cannot model the valuation as $\mathcal{V} \rightarrow \Lambda$.

Secondly, by modeling an argument as $\mathcal{V} \times \Lambda$, we allow the Cartesian product of $\mathcal{V}_{Lab} \times \Lambda$. We assume that the input specification is well-typed. Therefore the excessive introduced variables are harmless, as they are not present in the input mCRL2 specification.

Thirdly, we assume that all semantic interpretations evaluate to a well defined value, i.e., not $\perp$. If we allow the evaluation of $\perp$, it would pose all kind of problems, e.g., when communicating values. Hence, $\perp$ may never occur during simulation.

9.2.5 Data Expressions

Data expressions describe functions on syntactic variables. Data expressions are used to express action data parameters, process parameter updates and conditional choices. Internally, data expressions are specified through abstract data types. When we write a value, the value can internally represented by an application of constructor

functions and variables, e.g., the natural number 2 can internally be represented as $successor(successor(zero))$. Here, $successor(n)$ and $zero$ are constructor functions for the built-in sort $\mathbb{N}$. Like $\mathbb{N}$, other (basic) data types such as $\mathbb{Z}$, $\mathbb{B}$, *List*, *Set*, ... are part of the mCRL2 language.

Data expressions of the meta notation are modeled in a similar fashion as data expressions of the mCRL2 language. Hence, data expressions can be modeled by variables and functions (possibly) having arguments. For readability, presentation and modeling purposes, we also allow values as they provide elegant shorthand notations for compounded terms that consist of constructor functions and variables. Moreover, this allows for the conversion of meta notation data expressions into mCRL2 data expressions, which is shown at the end of this subsection.

When we assume that a structured sort $\mathcal{F}$ models the function symbols, we specify the data expression sort $\mathcal{E}$ as:

$$
\begin{aligned}
\textbf{sort} \quad \mathcal{E} = \textbf{struct}\ & \mathcal{E}_{\mathcal{V}}(dvr{:}\mathcal{V})?is_{\mathcal{E}}^{\mathcal{V}} \\
& \mid \mathcal{E}_{\Lambda}(dvl{:}\Lambda)?is_{\mathcal{E}}^{\Lambda} \\
& \mid \mathcal{E}_{expr}^{0}(f:\mathcal{F})?is_{\mathcal{E}}^{expr_0} \\
& \mid \mathcal{E}_{expr}^{1}(f:\mathcal{F}, expr_1{:}\mathcal{E})?is_{\mathcal{E}}^{expr_1} \\
& \mid \quad \vdots \qquad\qquad\qquad \ddots \\
& \mid \mathcal{E}_{expr}^{n-1}(f:\mathcal{F}, expr_1{:}\mathcal{E}, \ldots, expr_{n-1}{:}\mathcal{E})?is_{\mathcal{E}}^{expr_{n-1}} \\
& \mid \mathcal{E}_{expr}^{n}(f:\mathcal{F}, expr_1{:}\mathcal{E}, \ldots, expr_{n}{:}\mathcal{E})?is_{\mathcal{E}}^{expr_{n}};
\end{aligned}
$$

where a syntactic (typed) variable is modeled by $\mathcal{E}_{\mathcal{V}}$, a syntactic (typed) value is modeled by $\mathcal{E}_{\Lambda}$, and a syntactic (typed) function, with function symbol $f \in \mathcal{F}$ having arity i, is modeled by $\mathcal{E}_{expr}^{i}$. Example 9.5 and Example 9.6 respectively illustrate how variable and value data expressions are modeled in the meta notation.

Example 9.5(Data expression variable). Assume that a Boolean variable b occurs as a data expression. To model the expression in the meta notation, we write $\mathcal{E}_{\mathcal{V}}(\mathcal{V}_{\mathbb{B}}(b))$. △

Example 9.6(Data expression value). Assume that the Natural value 2 occurs as a data expression. To model the expression in the meta notation, we write $\mathcal{E}_{\Lambda}(\Lambda_{\mathbb{N}}(2))$. △

Data Expression Functions and Function Operators

A function operator describes either a label of a modeled constructor function or a label of a modeled mapping (function). Like variables and values, function operators are typed as well. Function operators are defined by the sort $\mathcal{F}$. Function operators are typed in a similar way as we have seen with values or variables. The structure sort $\mathcal{F}$ is constructed from an operator label sort $\mathcal{O}$ that defines the operation:

$$
\begin{aligned}
&\textbf{sort} \quad \mathcal{O} = \textbf{struct}\ O_1 \mid \ldots \mid O_n; \\
&\textbf{sort} \quad \mathcal{F} = \textbf{struct}\ Sort_{\mathcal{O}}^{1}(\pi_{op}{:}\mathcal{O})?is_{Sort_1} \mid \ldots \mid Sort_{\mathcal{O}}^{n}(\pi_{op}{:}\mathcal{O})?is_{Sort_n};
\end{aligned}
$$

Example 9.7(Data expression constructor functions). This example shows the modeling of constructor functions. We model a sort *GrayScale* that represents different gray scale levels:

sort *GrayScale*;

To model the whiteness level, we assume the constructor functions *white* and *darker*. In the original mCRL2 specification, they are modeled as:

cons *white* : *GrayScale*;
darker : *GrayScale* → *GrayScale*;

Here, the constructor function *white* has no arguments and the constructor function *darker* has one argument. To model the constructor functions in the meta notation, we first declare the following sorts:

sort $\mathcal{O} =$ **struct** *darker* | *white*;
sort $\mathcal{F} =$ **struct** $GrayScale_{\mathcal{O}}(\pi_{op}{:}\mathcal{O})?is_{GrayScale}$;

Subsequently, we model a data expression variable v that belongs to the sort *GrayScale*:

sort $\mathcal{V}_{Lab} =$ **struct** v;
sort $\mathcal{V} =$ **struct** $GrayScale_{\mathcal{V}}(\mathcal{V}_{Lab})$;

Now we can express:

- a *GrayScale* variable by: $\mathcal{E}_{\mathcal{V}}(GrayScale_{\mathcal{V}}(v))$.
- the *white* constructor, written in mCRL2 as *white* : *GrayScale*, is modeled by $\mathcal{E}^0_{expr}(GrayScale_{\mathcal{O}}(white))$. Because *white* is a constructor function that has a zero-arity, we use $\mathcal{E}^0_{expr}$.
- the *darker* constructor, written in mCRL2 as *darker* : *GrayScale* → *GrayScale* is modeled by $\mathcal{E}^1_{expr}(GrayScale_{\mathcal{O}}(darker), \mathcal{E}_{\mathcal{V}}(GrayScale_{\mathcal{V}}(v)))$. Since *darker* is a constructor function that has an arity of one, we use $\mathcal{E}^1_{expr}$.

△

Semantic Interpretation of Data Expressions

The semantic interpretation of a data expression is represented by a value. To compute a value we introduce the function $sem_{\mathcal{E}} : \mathcal{E} \times \mathcal{S} \rightarrow \Lambda$ that requires a data expression and a valuation. For the interpretation of data expression variables and data expression values we define the following data equations:

map $sem_{\mathcal{E}}{:}\mathcal{E} \times \mathcal{S} \rightarrow \Lambda$;
var $vr{:}\mathcal{V}$;
$vl{:}\Lambda$;
$\sigma{:}\mathcal{S}$;
$arg : Argument$;
eqn $sem_{\mathcal{E}}(\mathcal{E}_{\mathcal{V}}(vr), [\,]) = \bot$;
$sem_{\mathcal{E}}(\mathcal{E}_{\mathcal{V}}(vr, arg \triangleright \sigma) = if(\sigma(vr) \approx \mathcal{V}(arg), \pi_{\Lambda}(arg), sem_{\mathcal{E}}(\mathcal{E}_{\mathcal{V}}(vr), \sigma))$;
$sem_{\mathcal{E}}(\mathcal{E}_{\Lambda}(vl), \sigma) = vl$;

Data expression functions require separate data equations, which extend the ones from above. We only add the equations for the operators that are defined in the input mCRL2 specification. Let f be an operator with the result sort $Sort_f$, and let $\boxplus$ be a function, then for the defined functions we introduce:

map $sem_{\mathcal{E}} : \mathcal{E} \times \mathcal{S} \rightarrow \Lambda$;
var $expr_1, \ldots, expr_n{:}\mathcal{E}$;
$\sigma{:}\mathcal{S}$;
eqn $sem_{\mathcal{E}}(\mathcal{E}^n_{expr}(f, expr_1, \ldots, expr_n), \sigma) =$
$Sort_f(sem_{\mathcal{E}}(expr_1, \sigma) \boxplus \cdots \boxplus sem_{\mathcal{E}}(expr_n, \sigma))$;

Example 9.8(Semantic interpretation of a function). To demonstrate the semantic interpretation of a function, we model the semantic equivalence of two data expressions, modeled by the $\bowtie$ operator. The resulting sort of the function is the predefined sort $\mathbb{B}$, that needs to be converted to the meta notation sort $\mathbb{B}_{\Lambda}$. Since $\approx$ is defined for *all* sorts, the semantic interpretation function is modeled as:

var $expr_1, expr_2{:}\mathcal{E}$;
eqn $sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathcal{E}_{\mathcal{O}}(\bowtie), expr_1, expr_2), \sigma) = \mathbb{B}_{\Lambda}(sem_{\mathcal{E}}(expr_1, \sigma) \approx sem_{\mathcal{E}}(expr_2, \sigma)))$;

$\triangle$

Converting Meta Notation Values

For every modeled sort in the meta notation it is possible to specify all of its corresponding rewrite rules separately. From the experiences of the μCRL toolset and the motivation for developing its successor mCRL2 [GMWU06] show that specifying the rewrite rules for commonly accepted sorts, turns out to be annoying and a time consuming task. For practical reasons, we like to reuse rewrite rules that are either defined by the mCRL2 language or are provided in the translated specification. To circumvent the (re)specification in the meta notation, we provide converts from the meta notation values to values in the mCRL2 language. This is accomplished by the function $Sort_{\downarrow}{:}\Lambda \rightarrow Sort$, where $Sort \in \mathcal{S}^{\text{mCRL2}}$ denotes a sort in the mCRL2 specification. After a value has been converted to the mCRL2 language, it is possible to use the equations of the mCRL2 specification/language. To convert an mCRL2 value back to the meta notation, we use the appropriate constructor from sort Λ. Example 9.9 shows how to access mCRL2's built-in functionality, using the converts.

Example 9.9(Value conversion). This example expresses the addition on natural numbers in the meta notation using converts. We assume that sort $\mathbb{N}$ is modeled in the meta notation and the addition is defined by $\oplus$. We specify $\mathcal{O}$ and $\mathcal{F}$ as:

sort $\mathcal{O} = \textbf{struct}\ \oplus;$
sort $\mathcal{F} = \textbf{struct}\ \mathbb{N}_{\mathcal{O}}(\pi_{op}{:}\mathcal{O})?is_{\mathbb{N}};$

where an appropriate name is chosen for the projection function. The addition of the value 3 with the value 2 (i.e., the data expression $3+2$) is modeled as:

$$\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\oplus),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(3)),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(2)))$$

The sort $\mathbb{N}$ is a built-in sort that is accompanied with a set of rewrite rules including an operator for addition. We decide to use the internal rewrite rules. For that we convert sort $\mathbb{N}_{\Lambda}$ to $\mathbb{N}$ with the help of function $\mathbb{N}_{\downarrow}{:}\Lambda \rightarrow \mathbb{N}$:

map $\mathbb{N}_{\downarrow}{:}\Lambda \rightarrow \mathbb{N};$
var $n{:}\mathbb{N};$
eqn $\mathbb{N}_{\downarrow}(\mathbb{N}_{\Lambda}(n)) = n;$

Using the convert function $\mathbb{N}_{\downarrow}$ we specify the addition as:

var $expr_1, expr_2{:}\mathcal{E};$
eqn $sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathcal{E}_{\mathcal{O}}(\oplus), expr_1, expr_2), \sigma) =$
$\mathbb{N}_{\Lambda}(\mathbb{N}_{\downarrow}(sem_{\mathcal{E}}(expr_1, \sigma)) + \mathbb{N}_{\downarrow}(sem_{\mathcal{E}}(expr_2, \sigma))));$

Now, if we compute the semantic interpretation for some σ for

$$sem_{\mathcal{E}}(\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\oplus),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(3)),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(2))), \sigma)$$

the expression rewrites to the meta notation value $\mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(5))$. △

9.2.6 Multi-actions

The mCRL2 formalism defines syntactic multi-actions and semantic multi-action equivalence classes. Syntactic multi-actions are written in an mCRL2 specification. A syntactic multi-action consists of a list of syntactic actions. Semantic multi-action equivalence classes are observed during the execution of a process. A semantic multi-action equivalence class consists of an (ordered) list of semantic actions. This subsection explains the modeling of multi-actions in the meta notation, and how they are transformed from the syntactic to the semantic domain.

Syntactic Actions

The structured sort $Act_{\sqsubseteq}$ models a syntactic action, defined through two constructor functions. The first constructor function *Act* defines an external syntactic action,

i.e., an action that consists of an action label and an (optional) list of data parameters, written as a list of data expressions. The action label and the list of arguments are respectively retrieved with the help of the projection functions $\pi_{a_{lab}}{:}Act_{Lab}$ and $\pi_{args}{:}List(\mathcal{E})$. The second constructor function *tau* defines the internal syntactic action, i.e., τ.

sort $Act_{\Xi} = \textbf{struct}\ Act(\pi_{a_{lab}}{:}Act_{Lab}, \pi_{args}{:}List(\mathcal{E})) \mid tau;$

Semantic Actions

The sort Act_{Σ} models a semantic action, having one constructor function *ActSem* that corresponds to an externally observable semantic action. Multi-action equivalence classes that consist of only internal actions are modeled by empty lists. Therefore no constructor function is provided for an internal semantic action. We specify a semantic action as:

sort $Act_{\Sigma} = \textbf{struct}\ ActSem(\pi_{a_{lab}}{:}Act_{Lab}, \pi_{args}{:}List(\Lambda));$

Transforming Syntactic Actions into Semantic Actions

Syntactic actions are transformed into semantic actions with the help of function $sem_{Act_{\Xi}}$. The rewrite rule that defines the function is:

map $sem_{Act_{\Xi}}{:}Act_{\Xi} \times \mathcal{S} \rightarrow Act_{\Sigma};$
var $\sigma{:}\mathcal{S};$
$a{:}Act_{Lab};$
$args{:}List(\mathcal{E});$
eqn $sem_{Act_{\Xi}}(Act(a, args), \sigma) = ActSem(a, sem_{\mathcal{E}}^{List}(args, \sigma));$

The data equation for $sem_{Act_{\Xi}}(tau)$ is intentionally left unspecified, since the syntactic multi-action interpretation function removes all *tau*[§].

The semantic interpretation for a set of action data parameters (i.e., a list of data expressions) is computed by the function $sem_{\mathcal{E}}^{List}{:}List(\mathcal{E}) \times \mathcal{S} \rightarrow List(\Lambda)$:

map $sem_{\mathcal{E}}^{List}{:}List(\mathcal{E}) \times \mathcal{S} \rightarrow List(\Lambda);$
var $\sigma{:}\mathcal{S};$
$ds{:}List(\mathcal{E});$
$d{:}\mathcal{E};$
eqn $sem_{\mathcal{E}}^{List}([\,], \sigma) = [\,];$
$sem_{\mathcal{E}}^{List}(d \triangleright ds, \sigma) = sem_{\mathcal{E}}(d, \sigma) \triangleright sem_{\mathcal{E}}^{List}(ds, \sigma);$

Syntactic Multi-actions

The meta notation represents a syntactic multi-action $List(Act_{\Xi})$ as an unordered list of syntactic actions.

[§]The multi-action interpretation function is provided later in this section.

Semantic Multi-action Equivalence Classes

The meta notation represents the semantic multi-action equivalence class $List(Act_\Sigma)$ as an (ordered) list of semantic actions. Hidden actions are excluded from the list. Hence, the empty list represents the semantic equivalence class that consists of only the internal actions.

Transforming Syntactic Multi-actions into Multi-action equivalence classes

A syntactic [semantic] multi-action consists of syntactic [semantic] actions, and an action consists of a label and a list of data expressions [values]. Definition 2.2.16 is implemented in function $sem^{List}_{Act_\Xi}$:$List(Act_\Xi) \times \mathcal{S} \rightarrow List(Act_\Sigma)$ which transforms a list of syntactic actions into a multi-action equivalence class (i.e., an ordered list of semantic actions):

map $sem^{List}_{Act_\Xi}$:$List(Act_\Xi) \times \mathcal{S} \rightarrow List(Act_\Sigma)$;
sem_{Act_Ξ}:$Act_\Xi \times \mathcal{S} \rightarrow Act_\Sigma$;
var as:$List(Act_\Xi)$;
a:Act_Ξ;
σ:$\mathcal{S}$;
eqn $sem^{List}_{Act_\Xi}([], \sigma) = []$;
$sem^{List}_{Act_\Xi}(a \triangleright as, \sigma) = \; if(a \approx tau, sem^{List}_{Act_\Xi}(as, \sigma),$
$InsAct(sem_{Act_\Xi}(a, \sigma), sem^{List}_{Act_\Xi}(as, \sigma)));$

The insertion of a semantic action is performed by the auxiliary function $InsAct$:$Act_\Sigma \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$:

map $InsAct$:$Act_\Sigma \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$;
var x, y:Act_Σ;
ys:$List(Act_\Sigma)$;
eqn $InsAct(x, []) = [x]$;
$InsAct(x, y \triangleright ys) = if(x \leq y, x \triangleright y \triangleright ys, y \triangleright InsAct(x, ys))$;

Discussion

Multi-actions are mathematically a bag of actions. Therefore it would be natural to model them as such. The characterization of bags in the mCRL2 language are represented by functions with function updates that model exceptions. Since function sorts cannot be enumerated, bags cannot either. Because this functionality is required when transforming the syntactic multi-actions into semantic multi-actions (Example 9.10), we model them as a list of (un)ordered actions.

Example 9.10(Multi-actions represented by bags). If we model multi-actions as a bag of actions we require a function, say f, with the following signature:

$$Bag(Act_\Xi) \times \mathcal{S} \rightarrow Bag(Act_\Sigma)$$

that interprets a bag of syntactic multi-actions and transforms them into a new bag of semantic multi-actions. Ideally, f is performed as a map on each of the elements of the bag. Since we deal with infinite bags, we cannot apply f to individual elements. Instead, we need to define the set comprehension via the inverse of f, i.e., f^{-1}. This requires that the relation is a bijection. As the semantic interpretation function is not a bijection, we cannot define the inverse. $\triangle$

9.2.7 Transition Relation Representation

A state in an mCRL2 model is represented by a process term p and a data valuation σ. The meta notation introduces process X with a process parameter that stores the current value of the (counterpart) p and the value for the (counterpart) valuation σ. Process X describes the transition relation for a state (p,σ), i.e., $(p,\sigma) \xrightarrow{a} (p',\sigma')$, for which we (i) model the signature of the transition relation $\xrightarrow{a}$, and (ii) model the possible transitions along with the corresponding updated state (p',σ').

Data expressions cannot be directly used as transitions. To meet (i) we model the semantic multi-action equivalence class as a data parameter of the action $\mathbb{A}$. The action label denotes the kind of relation, i.e., the transition relation. To model the signature of $\xrightarrow{a}$ we declare (and use) the following mCRL2 action declaration:

act $\mathbb{A}{:}List(Act_\Sigma);$

To model (ii) we require a function that given a state, computes all transitions with the corresponding state updates for every transition relation. We introduce the sort $\mathcal{R}_{at}$ that models a triple of a multi-action, a process term and a data valuation by:

sort $\mathcal{R}_{at} = \textbf{struct}\ at(\pi_{ac}{:}List(Act_\Sigma), \pi_{p'}{:}\mathcal{P}, \pi_{\sigma'}{:}\mathcal{S});$

Here, at is the constructor function for a relation, argument π_{ac} denotes the multi-action equivalence class, argument $\pi_{p'}$ denotes the updated process term and argument $\pi_{\sigma'}$ denotes the updated data valuation. Every translated deduction rule d introduces a separate function $R_d{:}\mathcal{P}\times\mathcal{S}\rightarrow Set(\mathcal{R}_{at})$. All relations are computed by $R{:}\mathcal{P}\times\mathcal{S}\rightarrow Set(\mathcal{R}_{at})$, which specifies the union over the transition relations of the individual deduction rules.

map $R, R_{Alpha}, R_{Alt_1}, \ldots, R_{Def_1}, R_{Def_2}{:}\mathcal{P}\times\mathcal{S}\rightarrow Set(\mathcal{R}_{at});$
var $p{:}\mathcal{P};$
$s{:}\mathcal{S};$
eqn $R(p,s) = R_{Alpha}(p,s)\cup R_{Alt_1}(p,s)\cup\ldots\cup R_{Def_1}(p,s)\cup R_{Def_2}(p,s);$

Because only the applicable functions return a non-empty set, only the action transitions that can be performed remain. The implementation of the individual deduction rules can be found in Chapter 9.3. Thus we model the LPE as:

proc $X(p{:}\mathcal{P}, s{:}\mathcal{S}) = \sum_{r:\mathcal{R}_{at}} r\in R(p,s)\rightarrow \mathbb{A}(\pi_{ac}(r))\cdot X(\pi_{p'}(r), \pi_{\sigma'}(r))$

To initialize the LPS we write

init $X(p_0, \sigma_0)$;

where p_0, σ_0 denotes the initial state of the counterpart mCRL2 model. Initially we assume σ_0 to be empty.

9.3 Modeling Deduction Rules

This section describes the data equations that correspond to the deduction rules from the mCRL2 language. The assumptions and design decisions regarding the implementation are explicitly stated. The untimed deduction rules originate from Tables 2.1, 2.2, 2.4, 2.5, 2.6 and 2.7 (Chapter 2) and are modeled under the assumption that $\surd$ is replaced by the special $\surd_{\mathrm{p}}$ (Chapter 9.2.2).

9.3.1 Deadlock

A deadlock in an mCRL2 specification is modeled as the process term δ. The meta notation uses the expression *Deadlock*. Because the term has no deduction rules, we are not required to model data equations.

9.3.2 Multi-actions

When a syntactic multi-action α is performed the model performs an (observable) set of actions. In the meta notation we write a syntactic multi-action as:

$$Alpha([a_1, \ldots, a_n])$$

where $a_1, \ldots, a_n \in Act_{\Xi}$.

The function R_α computes the relations by a set comprehension, that correspond to the deduction rule of a multi-action (Table 2.1, rule *ma*), given a process term p and a valuation s. We allow a relation r in the set iff:

- the input term p is an action process term ($is_\alpha(p)$),
- the semantic multi-action corresponds to the syntactic multi-action evaluated under the data valuation ($\pi_{ac}(r) \approx sem^{List}_{Act_{\Xi}}(\pi_{multiaction}(p), s)$),
- the process denotes a successful termination ($is_\surd(\pi_{p'}(r))$, and
- the data valuation remains unchanged ($\pi_{\sigma'}(r) \approx s$).

With the help of the above conditions we express the corresponding data equation as:

eqn $R_\alpha(p, s) = if(is_\alpha(p), \{r{:}\mathcal{R}_{at} \mid \pi_{ac}(r) \approx sem^{List}_{Act_{\Xi}}(\pi_{multiaction}(p), s)$
$\wedge\ is_\surd(\pi_{p'}(r)) \ \wedge\ \pi_{\sigma'}(r) \approx s\}, \emptyset)$;

9.3.3 Alternative Operator

An alternative composition $p+q$ allows a non-deterministic choice when both process p as well as process q can perform a transition. In the meta notation we write the alternative composition as:

$$Alt(p,q)$$

where $p,q \in \mathcal{P}$ are process terms in the meta notation.

The deduction rules are provided in Table 2.1 by Alt_1, Alt_2, Alt_3 and Alt_4. The corresponding relations are respectively computed by the functions R_{Alt_1}, R_{Alt_2}, R_{Alt_3} and R_{Alt_4}. When a process term p or a process term q can perform a transition, respectively modeled by $R(\pi_1(p),s)$ and $R(\pi_2(p),s)$, then $R_{Alt_i}(p,s)$ $(1 \leq i \leq 4)$ can also perform a transition. Note, that the functions R_{Alt_1} and R_{Alt_3} explicitly state $\pi_{\sigma'}(r) \approx s$. These conjuncts are required for the deduction rules that replace $\checkmark$ by $\checkmark_{\mathbf{p}}$, because we assume that the data valuations remain unchanged.

eqn
$$\begin{aligned}
&R_{Alt_1}(p,s) = \mathit{if}(is_{Alt}(p), \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge is_{\checkmark}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset);\\
&R_{Alt_2}(p,s) = \mathit{if}(is_{Alt}(p), \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge \neg is_{\checkmark}(\pi_{p'}(r))\}, \emptyset);\\
&R_{Alt_3}(p,s) = \mathit{if}(is_{Alt}(p), \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_2(p),s) \wedge is_{\checkmark}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset);\\
&R_{Alt_4}(p,s) = \mathit{if}(is_{Alt}(p), \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_2(p),s) \wedge \neg is_{\checkmark}(\pi_{p'}(r))\}, \emptyset);
\end{aligned}$$

9.3.4 Sequential Operator

A sequential composition is denoted by $p \cdot q$. In the meta notation we express the sequential composition as:

$$Seq(p,q)$$

where $p,q \in \mathcal{P}$ are process terms in the meta notation.

Deduction rule Seq_1 in Table 2.1 expresses the successful termination of p. Deduction rule Seq_2 in Table 2.1 expresses the continuation as $p' \cdot q$ after p has performed an action and does not successfully terminate. Note that R_{Seq_2} demands that the signature of the resulting process term is again a sequential composition $(is_{Seq}(\pi_{p'}(r)))$. Hence, we specify the following two data equations:

eqn
$$\begin{aligned}
&R_{Seq_1}(p,s) = \mathit{if}(is_{Seq}(p),\\
&\quad \{r{:}\mathcal{R}_{at} \mid at(\pi_{ac}(r), \checkmark_{\mathbf{p}}, \pi_{\sigma'}(r)) \in R(\pi_1(p),s)\\
&\qquad \wedge\ \pi_{p'}(r) \approx \pi_2(p) \wedge\ \pi_{\sigma'}(r) \approx s\}, \emptyset);\\
&R_{Seq_2}(p,s) = \mathit{if}(is_{Seq}(p),\\
&\quad \{r{:}\mathcal{R}_{at} \mid is_{Seq}(\pi_{p'}(r)) \wedge\ at(\pi_{ac}(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p),s)\\
&\qquad \wedge\ \pi_2(\pi_{p'}(r)) \approx \pi_2(p) \wedge\ \neg is_{\checkmark}(\pi_1(\pi_{p'}(r)))\}, \emptyset);
\end{aligned}$$

9.3.5 Conditional Choice

The conditional choices $c \rightarrow p$ and $c \rightarrow p \diamond q$ allow the execution of behavior w.r.t. the evaluated condition. The first operator only executes the behavior of process p when

data expression c evaluates to *true*. The second operator has two bodies, i.e., p and q, for which the first process p is only executed when c evaluates to *true*. The second process q is only executed when c evaluates to *false*. The first operator is modeled by:

$$Cond_1(c,p)$$

The second operator is modeled by:

$$Cond_2(c,p,q)$$

In both meta notations, $c \in \mathcal{E}$ is a data expression and $p,q \in \mathcal{P}$ are process terms.

Both operands contain a syntactic Boolean data expression that requires a semantic interpretation. With the help of function $sem_{\mathcal{E}}$, we compute the semantic value for $\pi_c(p)$ (the projection function C applied to process term p) under the data valuation $s \in \mathcal{S}$. To evaluate the condition we convert the condition with function $\mathbb{B}_{\downarrow}$. The condition is written inside the guard of the *if*-statement, instead of the body of the set comprehension. This notation circumvents infinite rewrite sequences. A more detailed explanation is given in Chapter 10.3.

The first operator $c \rightarrow p$ corresponds to the deduction rules $Cond_1$ and $Cond_2$ in Table 2.1. for which we provide two data equations:

$$\begin{aligned}
\textbf{eqn}\quad & R_{Cond1_1}(p,s) = \mathit{if}(is_{Cond_1}(p) \wedge \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s)),\\
& \qquad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge is_{\surd}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\},\emptyset);\\
& R_{Cond1_2}(p,s) = \mathit{if}(is_{Cond_1}(p) \wedge \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s)),\\
& \qquad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge \neg is_{\surd}(\pi_{p'}(r))\},\emptyset);
\end{aligned}$$

The second operator $c \rightarrow p \diamond q$ corresponds to the deduction rules $Cond_1$', ..., $Cond_4$' in Table 2.1, for which we provide four data equations:

$$\begin{aligned}
\textbf{eqn}\quad & R_{Cond2_1}(p,s) = \mathit{if}(is_{Cond_2}(p) \wedge \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s)),\\
& \qquad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge is_{\surd}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\},\emptyset);\\
& R_{Cond2_2}(p,s) = \mathit{if}(is_{Cond_2}(p) \wedge \mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s)),\\
& \qquad \{r{:}\mathcal{R}_{at} \mid r \in R(\pi_1(p),s) \wedge \neg is_{\surd}(\pi_{p'}(r))\},\emptyset);\\
& R_{Cond2_3}(p,s) = \mathit{if}(is_{Cond_2}(p) \wedge \neg\mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s))\\
& \qquad ,\{r{:}\mathcal{R}_{at} \mid r \in R(\pi_2(p),s) \wedge is_{\surd}(\pi_{p'}(r)) \wedge \pi_{\sigma'}(r) \approx s\},\emptyset);\\
& R_{Cond2_4}(p,s) = \mathit{if}(is_{Cond_2}(p) \wedge \neg\mathbb{B}_{\downarrow}(sem_{\mathcal{E}}(\pi_c(p),s))\\
& \qquad ,\{r{:}\mathcal{R}_{at} \mid r \in R(\pi_2(p),s) \wedge \neg is_{\surd}(\pi_{p'}(r))\},\emptyset);
\end{aligned}$$

9.3.6 Sum Operator

The sum operator $\sum_{v:D} p$ specifies the enumeration of values over a domain of sort D and assigns the values to variable v. Under the selected values, the execution of process p is performed. The sum operator is modeled as:

$$Sum(v,p)$$

where $v \in \mathcal{V}$ is a (typed) variable and $p \in \mathcal{P}$ is a process term.

Although it is illegal to write "*true* $\approx$ 2" in an mCRL2 specification, the meta notation data expression:

$$\mathcal{E}^2_{expr}(\mathbb{N}_{\mathcal{O}}(\text{“}\approx\text{”}),\ \mathcal{E}_{\Lambda}(\mathbb{B}_{\Lambda}(true)),\ \mathcal{E}_{\Lambda}(\mathbb{N}_{\Lambda}(2)))$$

is a valid mCRL2 data expression. Hence, we need to restrict the generated values to the proper domain with the domain restriction function $M_D : \mathcal{V} \times \Lambda \rightarrow \mathbb{B}$, modeling $w \in M_D$ of rules Sum_1 and Sum_2. If a model specifies the sorts $Sort_1, \ldots, Sort_n$, we model each disjunct as separate conjunct in the restriction function. The function returns *true* if the sort's value corresponds to the sort's variable. The data equations that correspond to function M_D are:

map $M_D{:}\mathcal{V} \times \Lambda \rightarrow \mathbb{B}$;
var $v{:}\Lambda$;
$w{:}\mathcal{V}$;
eqn $M_D(v,w) = (is_{Sort_1}(v) \wedge is_{Sort_1}(w)) \vee \ldots \vee (is_{Sort_n}(v) \wedge is_{Sort_n}(w))$;

We assume that the sort order for $\mathcal{V}$ and Λ are identical to the order in which the sorts are declared in the provided mCRL2 specification.

The sum operator possibly extends the data valuation. To ensure that arguments in the valuation preserve their order, we add a new argument to the ordered list using $InsArg : Argument \times \mathcal{S} \rightarrow \mathcal{S}$, i.e., $InsArg(argument(v,w),\sigma) \triangleq \sigma[v \mapsto w]$:

map $InsArg : Argument \times \mathcal{S} \rightarrow \mathcal{S}$;
var $x, y : Argument$;
$ys : \mathcal{S}$;
eqn $InsArg(x, []) = [x]$;
$InsArg(x, y \triangleright ys) = if(x \leq y, x \triangleright y \triangleright ys, y \triangleright InsArg(x, ys))$;

Enumeration over values is defined by an existential quantifier in the body of the set comprehension. Let $p \equiv Sum(v, p')$ describe a sum operator in the meta notation. To update the value that belongs to variable v in the data valuation s we use function *InsArg*. The enumeration variable is obtained through the projection function π_v applied to p, i.e., $\pi_v(p)$. To find the enumerated values that are valid for this variable we restrict the set of possible values by $M_D(\pi_v(p), v)$. Hence, we model Sum_1 from Table 2.2 as:

eqn $R_{Sum_1}(p,s) = if(is_{Sum}(p), \{r{:}\mathcal{R}_{at} \mid \pi_{\sigma'}(r) \approx s \wedge is_{\surd}(\pi_{p'}(r))$
$\wedge\ \exists_{v:\Lambda} M_D(\pi_v(p), v) \wedge$
$(at(\pi_{ac}(r), \pi_{p'}(r), Z) \in R(\pi_1(p), Z)$
whr $Z = InsArg(argument(\pi_v(p), v), s)$ **end**
$)\}, \emptyset)$;

To model Sum_2 in Table 2.2 we require three auxiliary functions:

- The function $GenFreshVar : \mathcal{V} \times \mathbb{N} \rightarrow \mathcal{V}$ generates a fresh variable. Fresh variables are prefixed with a label, i.e., the constructor v_γ is reserved for the *GenFreshVar*

function. In this way the nth generated fresh variable (starting at 1) is represented by $v_\gamma(n)$:

$$
\begin{array}{ll}
\textbf{map} & \mathit{GenFreshVar}{:}\mathcal{V} \times \mathbb{N} \to \mathcal{V}; \\
\textbf{var} & l{:}\mathcal{V}_{Lab}; \\
 & id{:}\mathbb{N}; \\
\textbf{eqn} & \mathit{GenFreshVar}(\mathit{Sort}^1_{\mathcal{V}}(l), id) = \mathit{Sort}^1_{\mathcal{V}}(v_\gamma(id)); \\
 & \quad\vdots \\
 & \mathit{GenFreshVar}(\mathit{Sort}^n_{\mathcal{V}}(l), id) = \mathit{Sort}^n_{\mathcal{V}}(v_\gamma(id));
\end{array}
$$

- The function $\mathit{VariableSubstInProcessTerm}{:}(\mathcal{V} \to \mathcal{V}) \times \mathcal{P} \to \mathcal{P}$ renames all variables in a process term according to the provided variable substitution function. Since its implementation is lengthy and straightforward, it is provided in Appendix B.5.1.

- The function $\mathit{GetHighestId}{:}\mathcal{S} \to \mathbb{N}$ computes the highest generated identifier value according to a data valuation. It returns 0 when no identifiers are found.

$$
\begin{array}{ll}
\textbf{map} & \mathit{GetHighestId}{:}\mathcal{S} \to \mathbb{N}; \\
 & \mathit{GetVarId}{:}\mathit{Argument} \to \mathbb{N}; \\
\textbf{var} & \mathit{fs}{:}\mathcal{S}; \\
 & a{:}\mathit{Argument}; \\
\textbf{eqn} & \mathit{GetHighestId}([\,]) = 0; \\
 & \mathit{GetHighestId}(a \triangleright \mathit{fs}) = \max(\mathit{GetVarId}(a), \mathit{GetHighestId}(\mathit{fs})); \\
 & \mathit{GetVarId}(a) = \mathit{if}(\mathit{is}_{v_\gamma}(v_L(\pi_{\mathcal{V}}(a))), \pi_{id}(v_L(\pi_{\mathcal{V}}(a))), 0);
\end{array}
$$

Deduction rule Sum_2 generates a fresh variable, computed by the outcome of function $\mathit{GenFreshVar}(\pi_v(p), \mathit{GetHighestId}(s) + 1)$. Because the outcome is required twice, it is assigned to the where-clause VAR. The variable-to-variable substitution function $\lambda_{v:\mathcal{V}}(v)[\pi_v(p) \mapsto \text{VAR}]$ provides the first argument for the function *VariableSubstInProcessTerm*. For the second argument we add $\mathit{argument}(\text{VAR}, v)$ to valuation s with the help of function *InsArg*, such that we compute $\mathit{InsArg}(\mathit{argument}(\text{VAR}, v), s)$. Hence we model deduction rule R_{Sum_2} as:

$$
\begin{array}{ll}
\textbf{eqn} & R_{\mathit{Sum}_2}(p, s) = \mathit{if}(\mathit{is}_{\mathit{Sum}}(p), \{r{:}\mathcal{R}_{at} \mid \neg \mathit{is}_{\checkmark}(\pi_{p'}(r)) \\
 & \qquad \wedge (\exists_{v:\Lambda} M_D(\pi_v(p), v) \\
 & \qquad \wedge\ r \in R(\mathit{VariableSubstInProcessTerm}(\\
 & \qquad\qquad \lambda_{v:\mathcal{V}}(v)[\pi_v(p) \mapsto \text{VAR}], \pi_1(p)), \\
 & \qquad\qquad \mathit{InsArg}(\mathit{argument}(\text{VAR}, v), s)))\} \\
 & \qquad \textbf{whr}\ \text{VAR} = \mathit{GenFreshVar}(\pi_v(p), \mathit{GetHighestId}(s) + 1)\ \textbf{end} \\
 & \qquad , \emptyset);
\end{array}
$$

9.3.7 Parallel Operator

The parallel composition $p \parallel q$ denotes the concurrent execution of the processes p and q. The meta notation expresses the composition as:

$$Par(p,q)$$

where $p,q \in \mathcal{P}$ are meta notated process terms.

The semantics is provided in Table 2.4. The deduction rules Par_5 to Par_7 merge the semantics multi-action equivalence classes from the premises into a new semantic multi-action equivalence class in the conclusion. To merge the ordered lists of semantic multi-actions we model the auxiliary function $MergeActionLists : List(Act_\Sigma) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$, i.e., $MergeActionLists(n,m) \stackrel{\triangle}{=} (n \mid m)_\sim$:

map $MergeActionLists : List(Act_\Sigma) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$;
var $x, y : Act_\Sigma$;
$xs, ys : List(Act_\Sigma)$;
eqn $MergeActionLists([\,],[\,]) = [\,]$;
$MergeActionLists([\,], xs) = xs$;
$MergeActionLists(xs, [\,]) = xs$;
$MergeActionLists(x \triangleright xs, y \triangleright ys) =$
$\quad if(x \leq y, x \triangleright MergeActionLists(xs, y \triangleright ys),$
$\quad\quad y \triangleright MergeActionLists(x \triangleright xs, ys))$;

With the help of function *MergeActionLists* the deduction rules Par_1 to Par_7 are modeled straightforwardly. The deduction rule Par_8 requires several auxiliary functions, which are subsequently explained.

eqn $R_{Par_1}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid$
$\quad at(\pi_{ac}(r), \surd_{\mathbf{p}}, s) \in R(\pi_1(p), s) \wedge \ \pi_{p'}(r) \approx \pi_2(p) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset)$;
$R_{Par_2}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid$
$\quad is_{Par}(\pi_{p'}(r)) \wedge \ at(\pi_{ac}(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)$
$\quad \wedge \neg is_{\surd}(\pi_1(\pi_{p'}(r))) \wedge \ \pi_2(\pi_{p'}(r)) \approx \pi_2(p)\}, \emptyset)$;
$R_{Par_3}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid$
$\quad at(\pi_{ac}(r), \surd_{\mathbf{p}}, s) \in R(\pi_2(p), s) \wedge \ \pi_{p'}(r) \approx \pi_1(p) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset)$;
$R_{Par_4}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid$
$\quad is_{par}(\pi_{p'}(r)) \wedge \ at(\pi_{ac}(r), \pi_2(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_2(p), s)$
$\quad \wedge \neg is_{\surd}(\pi_2(\pi_{p'}(r))) \wedge \ \pi_1(\pi_{p'}(r)) \approx \pi_1(p)\}, \emptyset)$;
$R_{Par_5}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid is_{\surd}(\pi_{p'}(r)) \wedge$
$\quad \exists_{t_1,t_2:List(Act_\Sigma)} at(t_1, \surd_{\mathbf{p}}, s) \in R(\pi_1(p), s) \wedge \ at(t_2, \surd_{\mathbf{p}}, s) \in R(\pi_2(p), s)$
$\quad \wedge MergeActionLists(t_1, t_2) \approx \pi_{ac}(r)\}, \emptyset)$;
$R_{Par_6}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid$
$\quad \exists_{r_1,r_2:\mathcal{R}_{at}} r_1 \in R(\pi_1(p), s) \wedge \ r_2 \in R(\pi_2(p), s)$
$\quad \wedge is_{\surd}(\pi_{p'}(r_1)) \wedge \ \neg is_{\surd}(\pi_{p'}(r_2))$
$\quad \wedge MergeActionLists(\pi_{ac}(r_1), \pi_{ac}(r_2)) \approx \pi_{ac}(r)$
$\quad \wedge \ \pi_{p'}(r) \approx \pi_{p'}(r_2) \wedge \ \pi_{\sigma'}(r) \approx \pi_{\sigma'}(r_2)\}, \emptyset)$;

$$
\begin{aligned}
R_{Par_7}(p,s) = \mathit{if}(\mathit{is}_{Par}(p), \{r{:}\mathcal{R}_{at} \mid \\
&\exists_{r_1,r_2:\mathcal{R}_{at}} r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
&\wedge \neg \mathit{is}_{\surd}(\pi_{p'}(r_1)) \wedge\ \mathit{is}_{\surd}(\pi_{p'}(r_2)) \\
&\wedge \mathit{MergeActionLists}(\pi_{ac}(r_1), \pi_{ac}(r_2)) \approx \pi_{ac}(r) \\
&\wedge\ \pi_{p'}(r) \approx \pi_{p'}(r_1) \wedge\ \pi_{\sigma'}(r) \approx \pi_{\sigma'}(r_1)\}, \emptyset);
\end{aligned}
$$

To model deduction rule Par_8 from Chapter 9.2.1, we require nine auxiliary functions. These auxiliary functions are used to identify duplicate variables in the separate valuations, generate fresh variables and perform substitutions. The descriptions and implementations are provided next:

1. The function $\mathit{DuplicateVariablesInValuation}{:}\mathcal{S} \times \mathcal{S} \rightarrow \mathit{List}(\mathcal{V})$ takes two valuations and computes a list of overlapping variables, i.e., $\mathit{dom}(\sigma) \cap \mathit{dom}(\sigma')$. Hence, we model:

   map $\mathit{DuplicateVariablesInValuation} : \mathcal{S} \times \mathcal{S} \rightarrow \mathit{List}(\mathcal{V})$;
   var $x, y : \mathit{Argument}$;
   $xs : \mathcal{S}$;
   $ys : \mathcal{S}$;
   eqn $\mathit{DuplicateVariablesInValuation}([],ys) = []$;
   $\mathit{DuplicateVariablesInValuation}(xs,[]) = []$;
   $\mathit{DuplicateVariablesInValuation}(x \triangleright xs, y \triangleright ys) =$
   $\quad \mathit{if}(\pi_{\mathcal{V}}(x) \approx \pi_{\mathcal{V}}(y), \pi_{\mathcal{V}}(x) \triangleright \mathit{DuplicateVariablesInValuation}(xs,ys),$
   $\quad\quad \mathit{if}(\pi_{\mathcal{V}}(x) < \pi_{\mathcal{V}}(y), \mathit{DuplicateVariablesInValuation}(xs, y \triangleright ys),$
   $\quad\quad\quad \mathit{DuplicateVariablesInValuation}(x \triangleright xs, ys)));$

2. The function $\mathit{GenFreshVars}{:}\mathbb{N} \times \mathit{List}(\mathcal{V}) \rightarrow \mathit{List}(\mathcal{V})$ generates a list of fresh variables. It requires an identifier, i.e., a natural number, to generate unique variables. To assert that the freshly generated variables are properly typed, it requires a list of variables[¶]. The function *GenFreshVar* is described in Chapter 9.3.6.

   map $\mathit{GenFreshVars}{:}\mathbb{N} \times \mathit{List}(\mathcal{V}) \rightarrow \mathit{List}(\mathcal{V})$;
   var $vs{:}\mathit{List}(\mathcal{V})$;
   $v{:}\mathcal{V}$;
   $n{:}\mathbb{N}$;
   eqn $\mathit{GenFreshVars}(n,[]) = []$;
   $\mathit{GenFreshVars}(n, v \triangleright vs) = \mathit{GenFreshVar}(v,n) \triangleright \mathit{GenFreshVars}(n+1, vs)$;

3. The function $\mathit{GetHighestId}{:}\mathcal{S} \rightarrow \mathbb{N}$ has already been described in Chapter 9.3.6.

[¶]The freshly generated variables are later substituted for the ones provided here. It is important that the order of the variables (and therefore the sorts) stays preserved.

4. The function $CreateVariableSubst{:}List(\mathcal{V}) \times List(\mathcal{V}) \rightarrow (\mathcal{V} \rightarrow \mathcal{V})$ models $\overrightarrow{v} \rightarrow \overrightarrow{v'}$. It takes two variable lists and creates a variable substitution function. The first argument denotes a list of variables that is substituted. The second argument denotes a list of new variables. The lists must have the same length and the sorts need to be presented in the same order.

map $CreateVariableSubst{:}List(\mathcal{V}) \times List(\mathcal{V}) \rightarrow (\mathcal{V} \rightarrow \mathcal{V})$;
$CreateVariableSubst{:}List(\mathcal{V}) \times List(\mathcal{V}) \times (\mathcal{V} \rightarrow \mathcal{V}) \rightarrow (\mathcal{V} \rightarrow \mathcal{V})$;
var $x{:}\mathcal{V}$;
$x'{:}\mathcal{V}$;
$xs{:}List(\mathcal{V})$;
$xs'{:}List(\mathcal{V})$;
$\rho{:}\mathcal{V} \rightarrow \mathcal{V}$;
eqn $CreateVariableSubst(xs, xs') = CreateVariableSubst(xs, xs', \lambda v{:}\mathcal{V}.(v))$;
$CreateVariableSubst([\,], [\,], \rho) = \rho$;
$CreateVariableSubst(x \triangleright xs, x' \triangleright xs', \rho) =$
$\quad CreateVariableSubst(xs, xs', \rho[x \mapsto x'])$;

5. The function $VariableSubstInValuation{:}(\mathcal{V} \rightarrow \mathcal{V}) \times \mathcal{S} \rightarrow \mathcal{S}$ renames the variables in a data valuation for a given variable substitution function. Hence, we model $\sigma[\overrightarrow{v} \mapsto \overrightarrow{v'}]$ by $VariableSubstInValuation(\overrightarrow{v} \rightarrow \overrightarrow{v'}, \sigma)$.

map $VariableSubstInValuation{:}(\mathcal{V} \rightarrow \mathcal{V}) \times \mathcal{S} \rightarrow \mathcal{S}$;
var $\rho{:}\mathcal{V} \rightarrow \mathcal{V}$;
$fs{:}\mathcal{S}$;
$a{:}Argument$;
eqn $VariableSubstInValuation(\rho, [\,]) = [\,]$;
$VariableSubstInValuation(\rho, a \triangleright fs) =$
$\quad argument(\rho(\pi_{\mathcal{V}}(a)), \pi_{\Lambda}(a)) \triangleright VariableSubstInValuation(\rho, fs)$;

6. The function *VariableSubstInProcessTerm* is described in Chapter 9.3.6.

7. The function $ValuationMinusValuation{:}\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ takes two (ordered) valuations and removes the arguments from the first valuation if the variables also occur in the second valuation. Whenever we write $ValuationMinusValuation(\sigma, \sigma')$ it models $\sigma \setminus \sigma'$.

```
map   ValuationMinusValuation : S × S → S;
var   x, y : Argument;
      xs : S;
      ys : S;
eqn   ValuationMinusValuation([],ys) = [];
      ValuationMinusValuation(xs, []) = xs;
      ValuationMinusValuation(x ▷ xs, y ▷ ys) =
        if(x ≈ y, ValuationMinusValuation(xs,ys),
          if(x < y, x ▷ ValuationMinusValuation(xs, y ▷ ys),
            ValuationMinusValuation(x ▷ xs,ys)));
```

8. The function *MergeValuations* : $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ takes two (ordered) valuations and constructs a new (ordered) valuation. Thus *MergeValuations*(σ', σ') models $\sigma \cup \sigma'$.

```
map   MergeValuations : S × S → S;
var   x, y : Argument;
      xs,ys : S;
eqn   MergeValuations([], []) = [];
      MergeValuations([],xs) = xs;
      MergeValuations(xs, []) = xs;
      MergeValuations(x ▷ xs, y ▷ ys) =
        if(x ≤ y, x ▷ MergeValuations(xs, y ▷ ys),
          y ▷ MergeValuations(x ▷ xs,ys));
```

9. The function *MergeActionLists* is described at the start of this subsection.

For readability we construct a where-clause SUBST that models a variable substitution. Let $\pi_{\sigma'}(r_1)$ and $\pi_{\sigma'}(r_2)$ be the data valuations of respectively the premises on the left and right. Then the short-hand notation for the clause is defined through:

```
SUBST =  CreateVariableSubst(
         DUP, GenFreshVars(
         max(GetHighestId(π_σ'(r_1)), GetHighestId(π_σ'(r_2))) + 1, DUP)
         ) whr DUP  = DuplicateVariablesInValuation(
                      ValuationMinusValuation(π_σ'(r_2), s),
                      ValuationMinusValuation(π_σ'(r_1), s))
           end
```

With the help of the auxiliary functions we construct the data equation that belongs to deduction rule Par_8. Firstly, we compute the lowest identifier that can be used to generate fresh variables with max(*GetHighestId*$(\pi_{\sigma'}(r_1))$, *GetHighestId*$(\pi_{\sigma'}(r_2))) + 1$. Secondly, we compute the arguments that differ from the valuation s. The valuation of the left premise is computed by *ValuationMinusValuation*$(\pi_{\sigma'}(r_1), s)$. The valuation of the right premise is computed by *ValuationMinusValuation*$(\pi_{\sigma'}(r_2), s)$.

Thirdly, by using these two valuation, we compute the variables that have duplicate labels with function *DuplicateVariablesInValuation*. Because the outcome is required twice, we introduce a second where-clause DUP. Fourthly, with the help of $GenFreshVars(\max(GetHighestId(\pi_{\sigma'}(r_1)), GetHighestId(\pi_{\sigma'}(r_2))) + 1, \text{DUP})$, we generate a list of fresh variables that resolves the overlapping of variables. Finally, we compute the substitution function *CreateVariableSubst*. The result is assigned to the where-clause SUBST that is used in the functions *VariableSubstInProcessTerm* and *VariableSubstInValuation*. Hence, the data equation becomes:

$$
\begin{array}{ll}
\textbf{eqn} & R_{Par_8}(p,s) = if(is_{Par}(p), \{r{:}\mathcal{R}_{at} \mid \\
& \quad is_{Par}(\pi_{p'}(r)) \wedge \exists_{r_1,r_2:\mathcal{R}_{at}} \\
& \qquad (r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
& \qquad \wedge\ \neg is_{\surd}(\pi_{p'}(r_1)) \wedge\ \neg is_{\surd}(\pi_{p'}(r_2)) \\
& \qquad \wedge\ \pi_{ac}(r) \approx MergeActionLists(\pi_{ac}(r_1), \pi_{ac}(r_2)) \\
& \qquad \wedge\ \pi_1(\pi_{p'}(r)) \approx \pi_{p'}(r_1) \\
& \qquad \wedge\ \pi_2(\pi_{p'}(r)) \approx VariableSubstInProcessTerm(\text{SUBST}, \pi_{p'}(r_2)) \\
& \qquad \wedge\ \pi_{\sigma'}(r) \approx\ \ MergeValuations(\pi_{\sigma'}(r_1) \\
& \qquad\qquad\qquad ,VariableSubstInValuation(\text{SUBST}, \\
& \qquad\qquad\qquad\ \ ValuationMinusValuation(\pi_{\sigma'}(r_2),s))) \\
& \quad \textbf{whr}\ \ \text{SUBST} = \ldots\ \textbf{end}\ )\ \},\emptyset);
\end{array}
$$

9.3.8 Sync Operator

The sync composition $p \mid q$ denotes the synchronized execution of the first action from both process terms p and q, after which the remainder of the process term behaves concurrently. The meta notation that corresponds to $p \mid q$ is given by:

$$Sync(p,q)$$

where $p,q \in \mathcal{P}$ are meta notated process terms.

Deduction rules $Sync_1, \ldots, Sync_3$ in Table 2.5 and deduction rule $Sync_4$, discussed in Chapter 9.2.1, describe the semantics. The semantics is similar to the deduction rules for the parallel operator. Since we already discussed the design decisions for the unification of the valuations in Chapter 9.3.7, we only provide the (analogue) data equations. The implementation of the where-clause SUBST, modeled in the deduction rule $Sync_4$, is identical to the one modeled in the data equation R_{Par_8}.

$$
\begin{array}{ll}
\textbf{eqn} & R_{Sync_1}(p,s) = if(is_{Sync}(p), \{r : \mathcal{R}_{at} \mid \pi_{\sigma'}(r) \approx s \\
& \qquad \wedge\ is_{\surd}(\pi_{p'}(r)) \\
& \qquad \wedge\ \exists_{r_1,r_2:\mathcal{R}_{at}} r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
& \qquad \wedge\ is_{\surd}(\pi_{p'}(r_1)) \wedge\ is_{\surd}(\pi_{p'}(r_2)) \\
& \qquad \wedge\ MergeActionLists(\pi_{ac}(r_1), \pi_{ac}(r_2)) \approx \pi_{ac}(r)\},\emptyset);
\end{array}
$$

$$
\begin{aligned}
R_{Sync_2}(p,s) = \mathit{if}(\mathit{is}_{Sync}(p), \{r : \mathcal{R}_{at} \mid \\
& \exists_{r_1,r_2:\mathcal{R}_{at}} r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
& \wedge \neg \mathit{is}_{\surd}(\pi_{p'}(r_1)) \wedge\ \mathit{is}_{\surd}(\pi_{p'}(r_2)) \\
& \wedge \mathit{MergeActionLists}(\pi_{ac}(r_1), \pi_{ac}(r_2)) \approx \pi_{ac}(r) \\
& \wedge\ \pi_{p'}(r) \approx \pi_{p'}(r_1) \wedge\ \pi_{\sigma'}(r_1) \approx \pi_{\sigma'}(r)\}, \emptyset); \\
R_{Sync_3}(p,s) = \mathit{if}(\mathit{is}_{Sync}(p), \{r : \mathcal{R}_{at} \mid \\
& \exists_{r_1,r_2:\mathcal{R}_{at}} r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
& \wedge\ \mathit{is}_{\surd}(\pi_{p'}(r_1)) \wedge\ \neg \mathit{is}_{\surd}(\pi_{p'}(r_2)) \\
& \wedge \mathit{MergeActionLists}(\pi_{ac}(r_1), \pi_{ac}(r_2)) \approx \pi_{ac}(r) \\
& \wedge\ \pi_{p'}(r) \approx \pi_{p'}(r_2) \wedge\ \pi_{\sigma'}(r) \approx \pi_{\sigma'}(r_1)\}, \emptyset); \\
R_{Sync_4}(p,s) = \mathit{if}(\mathit{is}_{Sync}(p), \{r{:}\mathcal{R}_{at} \mid \\
& \mathit{is}_{Par}(\pi_{p'}(r)) \wedge \exists_{r_1,r_2:\mathcal{R}_{at}} \\
& \quad (r_1 \in R(\pi_1(p),s) \wedge\ r_2 \in R(\pi_2(p),s) \\
& \quad \wedge \neg \mathit{is}_{\surd}(\pi_{p'}(r_1)) \wedge\ \neg \mathit{is}_{\surd}(\pi_{p'}(r_2)) \\
& \quad \wedge\ \pi_{ac}(r) \approx \mathit{MergeActionLists}(\pi_{ac}(r_1), \pi_{ac}(r_2)) \\
& \quad \wedge\ \pi_1(\pi_{p'}(r)) \approx \pi_{p'}(r_1) \\
& \quad \wedge\ \pi_2(\pi_{p'}(r)) \approx \mathit{VariableSubstInProcessTerm}(\textsc{Subst}, \pi_{p'}(r_2)) \\
& \quad \wedge\ \pi_{\sigma'}(r) \approx\ \mathit{MergeValuations}(\pi_{\sigma'}(r_1) \\
& \qquad\qquad , \mathit{VariableSubstInValuation}(\textsc{Subst}, \\
& \qquad\qquad \mathit{ValuationMinusValuation}(\pi_{\sigma'}(r_2), s)))\}, \emptyset);
\end{aligned}
$$

9.3.9 Left Merge Operator

The left merge composition $p \lfloor\!\lfloor q$ expresses that the process term on the left has to perform an action first, before the remainder executes concurrently. In the meta notation, we write the composition as:

$$\mathit{Lmerge}(p,q)$$

assuming that $p, q \in \mathcal{P}$ are meta notation process terms.

The semantics is described by two deduction rules. The first rule Lmerge_1 in Table 2.5 expresses the successful termination of p after which the process behaves as q. The second rule Lmerge_2 expresses the continuation of $p' \parallel q$ after performing a first action from p. No explicit design decisions are taken. Hence the rules are straightforwardly modeled:

eqn

$$
\begin{aligned}
R_{Lmerge_1}(p,s) = \mathit{if}(\mathit{is}_{Lmerge}(p), \{r{:}\mathcal{R}_{at} \mid \mathit{at}(\pi_{ac}(r), \surd_{\mathbf{p}}, s) \in R(\pi_1(p),s) \\
\wedge\ \pi_{p'}(r) \approx \pi_2(p) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset); \\
R_{Lmerge_2}(p,s) = \mathit{if}(\mathit{is}_{Lmerge}(p), \{r{:}\mathcal{R}_{at} \mid \\
\mathit{is}_{Par}(\pi_{p'}(r)) \wedge\ \mathit{at}(\pi_{ac}(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p),s) \\
\wedge\ \pi_2(\pi_{p'}(r)) \approx \pi_2(p) \wedge\ \neg \mathit{is}_{\surd}(\pi_1(\pi_{p'}(r)))\}, \emptyset);
\end{aligned}
$$

9.3.10 Allow Operator

The allow term $\nabla_V(p)$ only permits the semantic multi-action equivalence classes for which the corresponding action labels are defined in the set of multi-action labels V

for p. The meta notation denotes the allow operator as:

$$Allow(V, p)$$

where $V{:}Set(Bag(Act_{Lab}))$ defines the set of permitted multi-action labels (represented by a bag of action labels) and $p \in \mathcal{P}$ defines the process term.

Deduction rules $Allow_1$ and $Allow_2$ in Table 2.6 describe the semantics that belong to the allow operator. We see that function $\underline{\alpha_{\sim}}$ (Definition 2.2.18) operates on inference rules for which it determines if the semantic multi-action equivalence class (without values) occurs in sets of semantic multi-action labels. To strip the data parameters from the actions, we specify the auxiliary function $actionlabels{:}List(Act_{\Sigma}) \rightarrow Bag(Act_{Lab})$:

map $actionlabels{:}List(Act_{\Sigma}) \rightarrow Bag(Act_{Lab})$;
var $ac{:}List(Act_{\Sigma})$;
$a{:}Act_{\Sigma}$;
eqn $actionlabels([]) = []$;
$actionlabels(a \triangleright ac) = \{\pi_{a_{lab}}(a){:}1\} \uplus actionlabels(ac)$;

Observe that the internal action ($\tau_{\sim}$) is always allowed by the allow operator. Hence, we extend the set of multi-action labels with the empty set. To ensure that the semantic multi-action equivalence class ac occurs in the set of allowed multi-actions labels $\pi_V(p) \cup \{\emptyset\}$, we state that the following condition must hold in the body of the set comprehension

$$actionlabels(ac) \in (\pi_V(p) \cup \{\emptyset\})$$

The remainder of the deduction rules is modeled straightforwardly:

eqn $R_{Allow_1}(p,s) = if(is_{Allow}(p), \{r{:}\mathcal{R}_{at} \mid is_{\surd}(\pi_{p'}(r)) \wedge\ r \in R(\pi_1(p), s)$
$\wedge\ actionlabels(\pi_{ac}(r)) \in (\pi_V(p) \cup \{\emptyset\}) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset)$;
$R_{Allow_2}(p,s) = if(is_{Allow}(p), \{r{:}\mathcal{R}_{at} \mid is_{Allow}(\pi_{p'}(r)) \wedge\ \neg is_{\surd}(\pi_1(\pi_{p'}(r)))$
$\wedge\ \pi_V(\pi_{p'}(r)) \approx \pi_V(p) \wedge\ at(\pi_{ac}(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)$
$\wedge\ actionlabels(\pi_{ac}(r)) \in (\pi_V(p) \cup \{\emptyset\})\}, \emptyset)$;

9.3.11 Block Operator

The block term $\partial_B(p)$ encapsulates all (multi)-actions for which an action label occurs in the set of blocking labels B performed by p. The block term is written in the meta notation as:

$$Block(B, p)$$

where $B{:}Set(Act_{Lab})$ is a set (of blocking) action labels and $p \in \mathcal{P}$ is the process term.

The deduction rules $Block_1$ and $Block_2$ are stated in Table 2.6. To determine if a part of an action label of a semantic multi-action equivalence class occurs in the set of blocking labels, we perform an abstraction on the class. The abstraction is provided through function *actionlabels*. The built-in function *Bag2Set* (transforms

the multi-action labels to a set of action labels), and the intersection of blocking labels. Only when the intersection between the blocking labels and the labels of a multi-action equivalence class are empty, it is possible to perform a transition. If $p \in \mathcal{P} \equiv Block(B, p')$ is a blocking process term, and if ac is the semantic multi-action equivalence class, then the following condition must hold in the body of the set comprehension:

$$Bag2Set(actionlabels(ac)) \cap (\pi_B(p)) \approx \emptyset$$

With the help of the auxiliary function we model the two deduction rules by:

eqn $R_{Block_1}(p,s) = if(is_{Block}(p), \{r{:}\mathcal{R}_{at} \mid is_{\surd}(\pi_{p'}(r))$
$\wedge\ r \in R(\pi_1(p),s) \wedge\ Bag2Set(actionlabels(\pi_{ac}(r))) \cap (\pi_B(p)) \approx \emptyset$
$\wedge \pi_{\sigma'}(r) \approx s\}, \emptyset);$
$R_{Block_2}(p,s) = if(is_{Block}(p), \{r{:}\mathcal{R}_{at} \mid is_{Block}(\pi_{p'}(r))$
$\wedge\ \neg is_{\surd}(\pi_1(\pi_{p'}(r))) \wedge\ \pi_B(\pi_{p'}(r)) \approx \pi_B(p)$
$\wedge\ at(\pi_{ac}(r), \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p),s)$
$\wedge\ Bag2Set(actionlabels(\pi_{ac}(r))) \cap (\pi_B(p)) \approx \emptyset\}, \emptyset);$

9.3.12 Action Rename Operator

The action rename term $\rho_R(p)$ renames (multi)-action labels according to function R (Definition 2.2.18) for a process term p. In the meta notation the action rename operator is written as:

$$Rename(Ren, p)$$

where $Ren{:}Act_{Lab} \rightarrow Act_{Lab}$ is a modeled rename function R and $p \in \mathcal{P}$ is a process term. The rename function *Ren* is modeled as an identity function *ID*, where function updates model the renaming for selective updates. The function *ID* is modeled as:

map $ID{:}Act_{Lab} \rightarrow Act_{Lab};$
var $x{:}Act_{Lab};$
eqn $ID(x) = x;$

Thus the renaming of $\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$ for *Ren* is then defined through $ID[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$.

To perform the actual renaming of the labels, i.e., $R \bullet (ac)$, we introduce function $Act_{Rename}{:}(Act_{Lab} \rightarrow Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$. The function requires an action label rename function and a semantic multi-action equivalence class, and produces a semantic multi-action in which the action labels are renamed and ordered simultaneously:

map $Act_{Rename}{:}(Act_{Lab} \rightarrow Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma);$
var $f{:}Act_{Lab} \rightarrow Act_{Lab};$
$a{:}Act_\Sigma;$
$ac{:}List(Act_\Sigma);$
eqn $Act_{Rename}(f, [\,]) = [\,];$
$Act_{Rename}(f, a \triangleright ac) = InsAct(ActSem(f(\pi_{a_{lab}}(a)), \pi_{args}(a)), Act_{Rename}(f, ac));$

Let $p \equiv Rename(Ren, p')$ be an action rename operator and let ac be a semantic multi-action, then $Act_{Rename}(\pi_{Ren}(p), ac)$ returns a semantic multi-action in which the action rename function $\pi_{Ren}(p)$ as been applied to ac. To find a valid substitution for a semantic multi-action equivalence class we introduce semantic multi-action ac'. The deduction rules Ren_1 and Ren_2 are then modeled as:

eqn $R_{Rename_1}(p,s) = if(is_{Rename}(p), \{r{:}\mathcal{R}_{at} \mid$
$is_{\surd}(\pi_{p'}(r)) \wedge \exists_{ac':List(Act_\Sigma)} \pi_{ac}(r) \approx Act_{Rename}(\pi_{Ren}(p), ac')$
$\wedge\ at(ac', \pi_{p'}(r), s) \in R(\pi_1(p), s) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset);$
$R_{Rename_2}(p,s) = if(is_{Rename}(p), \{r{:}\mathcal{R}_{at} \mid$
$\pi_{Ren}(\pi_{p'}(r)) \approx \pi_{Ren}(p) \wedge is_{Rename}(\pi_{p'}(r)) \wedge \neg is_{\surd}(\pi_1(\pi_{p'}(r)))$
$\wedge\ \exists_{ac':List(Act_\Sigma)} \pi_{ac}(r) \approx Act_{Rename}(\pi_{Ren}(p), ac')$
$\wedge\ at(ac', \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)\}, \emptyset);$

9.3.13 Hide Operator

The hide term $\tau_I(p)$ hides all actions in a semantic multi-action equivalence class, for which the corresponding label occurs in the set of labels I. The meta notation expresses this term as:

$$Hide(I, p)$$

where $I{:}Set(Act_{Lab})$ is the set of action labels and $p \in \mathcal{P}$ is a process term.

Hiding actions in a semantic multi-action equivalence class ac is performed by the function $Act_{Hide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$, thereby implementing $\theta_I(ac)$ from Definition 2.2.18, where I is a set of action labels:

map $Act_{Hide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma);$
var $I{:}Set(Act_{Lab});$
$a{:}Act_\Sigma;$
$ac{:}List(Act_\Sigma);$
eqn $Act_{Hide}(I, [\,]) = [\,];$
$Act_{Hide}(I, a \triangleright ac) = if(\pi_{a_{lab}}(a) \in I, Act_{Hide}(I, ac), a \triangleright Act_{Hide}(I, ac));$

Let p be $Hide(I, p')$ and let ac be a semantic multi-action equivalence class then $Act_{Hide}(\pi_I(p), ac)$ returns the semantic multi-action equivalence class in which the actions are hidden. Because the semantic actions are provided in an ordered list, removing an element preserves the order. Therefore it is not required to order the list afterwards.

Using function Act_{Hide} and an additional semantic multi-action ac' (to find a valid substitution), we specify the deduction rules $Hide_1$ and $Hide_2$ of Table 2.6 as:

eqn $R_{Hide_1}(p,s) = if(is_{Hide}(p), \{r{:}\mathcal{R}_{at} \mid$
$is_{\surd}(\pi_{p'}(r)) \wedge \exists_{ac':List(Act_\Sigma)}$
$\pi_{ac}(r) \approx Act_{Hide}(\pi_I(p), ac') \wedge\ at(ac', \pi_{p'}(r), s) \in R(\pi_1(p), s) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset);$
$R_{Hide_2}(p,s) = if(is_{Hide}(p), \{r{:}\mathcal{R}_{at} \mid$
$\pi_I(\pi_{p'}(r)) \approx \pi_I(p) \wedge is_{Hide}(\pi_{p'}(r)) \wedge \neg is_{\surd}(\pi_1(\pi_{p'}(r))) \wedge \exists_{ac':List(Act_\Sigma)}$
$\pi_{ac}(r) \approx Act_{Hide}(\pi_I(p), ac') \wedge\ at(ac', \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)\}, \emptyset);$

9.3.14 Prehide Operator

The prehide term $\Upsilon_U(p)$ prehides all actions for which the action label occurs in the set of prehiding labels. All action data parameters are removed and the actions are relabeled to *int* for only those actions for which the label occurs in U. In the meta notation the operator is written as:

$$Prehide(U, p)$$

where $U{:}Set(Act_{Lab})$ is the set of action labels that prehides the corresponding actions in process term $p \in \mathcal{P}$.

The function $Act_{Prehide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$ prehides actions in a semantic multi-action equivalence class ac. It models $\eta_U(ac)$ from Definition 2.2.18 where U is the set of action labels that are prehidden and ac is the semantic multi-action equivalence class. Note, that *int* is a reserved action label, that must be modeled by Act_{Lab}.

$$\begin{array}{ll}
\textbf{map} & Act_{Prehide}{:}Set(Act_{Lab}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma); \\
\textbf{var} & U{:}Set(Act_{Lab}); \\
 & ac{:}List(Act_\Sigma); \\
 & a{:}Act_\Sigma; \\
\textbf{eqn} & Act_{Prehide}(U, [\,]) = [\,]; \\
 & Act_{Prehide}(U, a \triangleright ac) = if(\pi_{a_{lab}}(a) \in U, InsAct(ActSem(int, [\,]), \\
 & \qquad\qquad Act_{Prehide}(U, ac)), a \triangleright Act_{Prehide}(U, ac));
\end{array}$$

With function $Act_{Prehide}$ we model the deduction rules Pre_1 and Pre_2 as:

$$\begin{array}{ll}
\textbf{eqn} & R_{Pre_1}(p, s) = if(is_{Prehide}(p), \{r{:}\mathcal{R}_{at} \mid is_{\surd}(\pi_{p'}(r)) \\
 & \qquad \wedge\ \exists_{ac'{:}List(Act_\Sigma)} \pi_{ac}(r) \approx Act_{Prehide}(\pi_U(p), ac') \\
 & \qquad \wedge\ at(ac', \pi_{p'}(r), s) \in R(\pi_1(p), s) \wedge \pi_{\sigma'}(r) \approx s\}, \emptyset); \\
 & R_{Pre_2}(p, s) = if(is_{Prehide}(p), \{r{:}\mathcal{R}_{at} \mid \pi_U(\pi_{p'}(r)) \approx \pi_U(p) \\
 & \qquad \wedge is_{Prehide}(\pi_{p'}(r)) \wedge\ \neg is_{\surd}(\pi_1(\pi_{p'}(r))) \wedge\ \exists_{ac'{:}List(Act_\Sigma)} \\
 & \qquad\quad \pi_{ac}(r) \approx Act_{Prehide}(\pi_U(p), ac') \wedge \\
 & \qquad\quad at(ac', \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)\}, \emptyset);
\end{array}$$

9.3.15 Communication Operator

The communication term $\Gamma_C(p)$ renames synchronizing actions. Actions are renamed when the bag of action labels occurs in the multi-action and the action data parameters all have the same semantic value. The communication is specified by a partial function, where $dom(C)$ denotes the bags of synchronizing action labels, and $range(C)$ specifies the result of the action label renaming. The data parameters remain unchanged during the synchronization. A communication mapping is modeled by the sort $\mathcal{C}$.

$$\textbf{sort} \quad \mathcal{C} = \textbf{struct}\ communication(C_{dom}{:}List(Act_{Lab}), C_{range}{:}Act_{Lab});$$

The bag of synchronizing action labels is specified through C_{dom}. The resulting action label is specified through C_{range}. The partial communication function is modeled by a list of $\mathcal{C}$. The communication term is modeled as:

$$Comm(C^{List}_{comm}, p)$$

where C^{List}_{comm}:$List(\mathcal{C})$ is a list of communications and p:$\mathcal{P}$ is a process term.

Function γ_C from Definition 2.2.18 is modeled by the function Act_{Comm}. The basic idea of the function is, that it first constructs a mapping which maps the value list from the data parameters to a bag of action labels from a multi-action equivalence class. The mapping is computed by function f^{d2a}. This initially maps all value lists to an empty bag of action labels. Secondly, we traverse the semantic multi-action equivalence class using $Act_{Comm'}$ for the first communication. If we encounter an action, for which the list of values maps to a bag of action labels that is a subset for the bag of communication labels, we substitute the matching actions with the communication result, and recompute for the remaining multi-action equivalence class its data value to action label mapping. If an action does not communicate, it is added to the list of remaining actions that need to be traversed by the next communication. The multi-action equivalence class is traversed until all actions have been inspected. Then it removes the communication and proceeds with the next communication, thereby using the list of remaining actions. The process is repeated until all communications are processed.

map Act_{Comm}:$List(\mathcal{C}) \times List(Act_\Sigma) \rightarrow List(Act_\Sigma)$;
var ac:$List(Act_\Sigma)$;
C_{comm}:$\mathcal{C}$;
C^{List}_{comm}:$List(\mathcal{C})$;
eqn $Act_{Comm}([], ac) = ac$;
$Act_{Comm}(C_{comm} \triangleright C^{List}_{comm}, ac) =$
$Act_{Comm'}(C_{comm}, C^{List}_{comm}, ac, f^{d2a}(ac, \lambda_{x:List(\Lambda)}\emptyset), L2B(C_{dom}(C_{comm})), [], [])$;

The function $f^{d2a} : List(Act_\Sigma) \times (List(\Lambda) \rightarrow Bag(Act_{Lab})) \rightarrow (List(\Lambda) \rightarrow Bag(Act_{Lab}))$ constructs a mapping $d2a : List(\Lambda) \rightarrow Bag(Act_{Lab})$ that relates lists of values to bags of action labels from a list of semantic actions:

map f^{d2a}:$List(Act_\Sigma) \times (List(\Lambda) \rightarrow Bag(Act_{Lab})) \rightarrow (List(\Lambda) \rightarrow Bag(Act_{Lab}))$;
var $d2a$:$List(\Lambda) \rightarrow Bag(Act_{Lab})$;
ac:$List(Act_\Sigma)$;
a:Act_Σ;
eqn $f^{d2a}([], d2a) = d2a$;
$f^{d2a}(a \triangleright ac, d2a) = f^{d2a}(ac, d2a[\pi_{args}(a) \mapsto d2a(\pi_{args}(a)) \uplus \{\pi_{a_{lab}}(a):1\}])$;

The bag of communicating action labels is computed by function $L2B$:

map $L2B{:}List(Act_{Lab}) \rightarrow Bag(Act_{Lab});$
var $a_{lab}{:}Act_{Lab}$
$c_{dom}{:}List(Act_{Lab})$
eqn $L2B([\,]) = \emptyset;$
$L2B(a_{lab} \triangleright c_{dom}) = \{a_{lab}{:}1\} \uplus L2B(c_{dom});$

We traverse a multi-action equivalence class by function $Act_{Comm'}$. If we find a match $(c_{comm} \subseteq d2a(\pi_{args}(a))))$, then the matching semantic actions are replaced by the synchronizing result. The substitution is performed by removing the matching actions, using function $Actions^{-}$. The result is added to the list successful communication actions $(InsAct(c_{comm}, ActSem(C_{range}(C_{comm}), \pi_{args}(a)), Act_{Result}))$. Then we recompute the lists of data values to action labels mapping $(f^{d2a}(Actions^{-}(C_{dom}(C_{comm}), a \triangleright ac, \pi_{args}(a)), \lambda_{x:List(\Lambda)}\emptyset))$. If the action does not match, it is added to the list of remaining actions $(Act_{Comm'}(C_{comm}, C^{List}_{comm}, ac, d2a, c_{comm}, Act_{Result}, a \triangleright Act_{Remain}))$.

map $Act_{Comm'}{:}\mathcal{C} \times List(\mathcal{C}) \times List(Act_{\Sigma}) \times (List(\Lambda) \rightarrow Bag(Act_{Lab}))$
$\times Bag(Act_{Lab}) \times List(Act_{\Sigma}) \times List(Act_{\Sigma}) \rightarrow List(Act_{\Sigma});$
var $d2a{:}List(\Lambda) \rightarrow Bag(Act_{Lab});$
$c_{comm}{:}Bag(Act_{Lab});$
$Act_{Result}, Act_{Remain}{:}List(Act_{\Sigma});$
$C_{comm}{:}\mathcal{C};$
$C^{List}_{comm}{:}List(\mathcal{C});$
$ac{:}List(Act_{\Sigma});$
$a{:}Act_{\Sigma};$
eqn $Act_{Comm'}(C_{comm}, C^{List}_{comm}, [\,], d2a, c_{comm}, Act_{Result}, Act_{Remain}) =$
$MergeActionLists(Act_{Result}, Act_{Comm}(C^{List}_{comm}, Act_{Remain}));$
$Act_{Comm'}(C_{comm}, C^{List}_{comm}, a \triangleright ac, d2a, c_{comm}, Act_{Result}, Act_{Remain}) =$
$if(c_{comm} \subseteq d2a(\pi_{args}(a)),$
$Act_{Comm'}(\ C_{comm}, C^{List}_{comm}, Actions^{-}(C_{dom}(C_{comm}), a \triangleright ac, \pi_{args}(a)),$
$f^{d2a}(Actions^{-}(C_{dom}(C_{comm}), a \triangleright ac, \pi_{args}(a)), \lambda_{x:List(\Lambda)}\emptyset),$
$InsAct(c_{comm}, ActSem(C_{range}(C_{comm}), \pi_{args}(a)), Act_{Result}),$
$Act_{Remain})$
$, Act_{Comm'}(C_{comm}, C^{List}_{comm}, ac, d2a, c_{comm}, Act_{Result}, a \triangleright Act_{Remain})$
$);$

Here, function $Actions^{-}$ is defined as:

map $Actions^{-}{:}List(Act_{Lab}) \times List(Act_{\Sigma}) \times List(\Lambda) \rightarrow List(Act_{\Sigma});$
var $a_{lab}{:}Act_{Lab};$
$c_{dom}{:}List(Act_{Lab});$
$ac{:}List(Act_{\Sigma});$
$args{:}List(\Lambda);$
eqn $Actions^{-}([\,], ac, args) = ac;$
$Actions^{-}(a_{lab} \triangleright c_{dom}, ac, args) =$
$Actions^{-}(c_{dom}, Action^{-}(ActSem(a_{lab}, args), ac), args);$

The function $Actions^-$ uses the auxiliary function $Action^-$ to remove individual actions from a list of semantic actions:

map $Action^-{:}Act_\Sigma \times List(Act_\Sigma) \rightarrow List(Act_\Sigma);$
var $a, b{:}Act_\Sigma;$
$ac{:}List(Act_\Sigma);$
eqn $Action^-(a, []) = [];$
$Action^-(a, b \triangleright ac) = if(a \approx b, ac, b \triangleright Action^-(a, ac));$

With the help of these functions we model the deduction rules $Comm_1$ and $Comm_2$ as:

eqn $R_{Comm_1}(p,s) = if(is_{Comm}(p), \{r{:}\mathcal{R}_{at} \mid is_{\surd}(\pi_{p'}(r))$
$\wedge\ \exists_{ac'{:}List(Act_\Sigma)} at(ac', \pi_{p'}(r), s) \in R(\pi_1(p), s) \wedge \pi_{\sigma'}(r) \approx s$
$\wedge\ \pi_{ac}(r) \approx Act_{Comm}(\pi_{C^{List}_{comm}}(p), ac')\}, \emptyset);$
$R_{Comm_2}(p,s) = if(is_{Comm}(p), \{r{:}\mathcal{R}_{at} \mid \pi_{C^{List}_{comm}}(\pi_{p'}(r)) \approx \pi_{C^{List}_{comm}}(p)$
$\wedge is_{Comm}(\pi_{p'}(r)) \wedge\ \neg is_{\surd}(\pi_1(\pi_{p'}(r))) \wedge\ \exists_{ac'{:}List(Act_\Sigma)}$
$\pi_{ac}(r) \approx Act_{Comm}(\pi_{C^{List}_{comm}}(p), ac')$
$\wedge\ at(ac', \pi_1(\pi_{p'}(r)), \pi_{\sigma'}(r)) \in R(\pi_1(p), s)\}, \emptyset);$

9.3.16 Process Definition

The mCRL2 language describes the set of process definitions as a system of process equations. An equation consists of a process label, a list of process parameters and a process expression. The system of process equations is defined by PE (Definition 2.2.10). A process definition is specified as $X(\overrightarrow{v}) = p$ where $X \in PE$ and a process reference is written as $X(\overrightarrow{v{=}d})$. We have chosen to model this syntax for its flexibility, i.e., the process parameter updates can be specified in random order, results in a concise and readable meta notation, and intuitively specifies the variable substitution inside a process term.

Process Equation System When $\{X_1, \ldots, X_n\}$ specifies the set of all process labels from an mCRL2 specification, then the sort $\mathcal{X}$ models the process labels:

sort $\mathcal{X} = \textbf{struct}\ X_1 \mid \ldots \mid X_n;$

To model PE we introduce a Process Equation System function $PES : \mathcal{X} \rightarrow \mathcal{P}$ that maps process labels to process terms. When we assume that $PE \equiv \{X_1(\overrightarrow{v_1}) = p_1, \ldots, X_n(\overrightarrow{v_n}) = p_n\}$, where $\overrightarrow{v_1}, \ldots, \overrightarrow{v_n}$ denote the process parameters, and $p_1, \ldots, p_n$ denote the associated process terms, then we model the equation system as:

sort $\mathcal{X} = \textbf{struct}\ X_1 \mid \ldots \mid X_n;$
map $PES{:}\mathcal{X} \rightarrow \mathcal{P}$
eqn $PES(X_1) = p_1;$
$\vdots$
$PES(X_n) = p_n;$

Process References The term $X(v_1{=}d_1,\ldots,v_n{=}d_n)$ expresses a process reference in the mCRL2 language. A process parameter update (or an assignment) $v_i{=}d_i, (1 \leq i \leq n)$ is modeled by sort $\mathcal{Q}$:

sort $\mathcal{Q} = \textbf{struct}\ \mathit{ProcParAss}(\pi_{\mathcal{V}}{:}\mathcal{V}, \mathit{dataexpression}{:}\mathcal{E});$

The process reference itself is modeled as:

$$\mathit{Def}(X, [\mathit{ProcParAss}(v_1, d_1), \ldots, \mathit{ProcParAss}(v_n, d_n)])$$

where $X \in \mathcal{X}$ is a process label, $v_i \in \mathcal{V}$ is a variable label and $d_i \in \mathcal{E}$ is a data expression, for $1 \leq i \leq n$. Using *Def* in conjunction with *PES* specifies a mechanism that assigns data expressions to local variables and provides a substitution for variables in both the process terms and data valuations.

Deduction Rules To model $\{[\overrightarrow{d}]\}^{\sigma}$ from deduction rule Def_1 in Table 2.7 and Def_2 from Chapter 9.2.1, we interpret the data expressions on the right hand-side of the assignments. Using function $\mathit{Assignments}^{\sigma}{:}\mathit{List}(\mathcal{Q}) \times \mathcal{S} \rightarrow \mathcal{S}$ we interpret the values and update the data valuation:

map $\mathit{Assignments}^{\sigma} : \mathit{List}(\mathcal{Q}) \times \mathcal{S} \rightarrow \mathcal{S};$
$\mathit{Assignments}^{\sigma} : \mathcal{Q} \times \mathcal{S} \rightarrow \mathit{Argument};$
var $p : \mathcal{Q};$
$pl : \mathit{List}(\mathcal{Q});$
$s : \mathcal{S};$
eqn $\mathit{Assignments}^{\sigma}(p,s) = \mathit{argument}(\pi_{\mathcal{V}}(p), \mathit{sem}_{\mathcal{E}}(\mathit{dataexpression}(p), s));$
$\mathit{Assignments}^{\sigma}([\,],s) = [\,];$
$\mathit{Assignments}^{\sigma}(p \triangleright pl, s) = \mathit{InsArg}(\mathit{Assignments}^{\sigma}(p,s), \mathit{Assignments}^{\sigma}(pl,s));$

For deduction rule Def_1 from Table 2.7 we model $\sigma[\overrightarrow{v} \mapsto \{[\overrightarrow{d}]\}^{\sigma}]$ by computing the process parameter updates with the function $\mathit{Assignments}^{\sigma}$ for the assignments $\pi_{\mathit{ProcParAsss}}(p)$ and the data valuation s. To update variables in a data valuation we subsequently remove duplicate variables with the function $\mathit{RemoveArgWithDupVar} : \mathit{List}(\mathcal{Q}) \times \mathcal{S} \rightarrow \mathcal{S}$. The auxiliary function $\mathit{RemoveArgWithDupVar}'$ requires an ordered list of assignments. Because assignments can be specified in any order, they are ordered by the function *OrderPP*:

map $\mathit{RemoveArgWithDupVar} : \mathit{List}(\mathcal{Q}) \times \mathcal{S} \rightarrow \mathcal{S};$
$\mathit{RemoveArgWithDupVar}' : \mathit{List}(\mathcal{Q}) \times \mathcal{S} \rightarrow \mathcal{S};$
$\mathit{OrderPP} : \mathit{List}(\mathcal{Q}) \rightarrow \mathit{List}(\mathcal{Q});$
$\mathit{InsertPP} : \mathcal{Q} \times \mathit{List}(\mathcal{Q}) \rightarrow \mathit{List}(\mathcal{Q});$
var $p, q : \mathcal{Q};$
$lp, lq : \mathit{List}(\mathcal{Q});$
$v : \mathit{Argument};$
$vl : \mathcal{S};$

eqn $OrderPP([]) = [];$
$OrderPP(p \triangleright lp) = InsertPP(p, OrderPP(lp));$
$InsertPP(p, []) = [p];$
$InsertPP(p, q \triangleright lq) = if(p \leq q, p \triangleright q \triangleright lq, q \triangleright InsertPP(p, lq));$
$RemoveArgWithDupVar(lp, vl) =$
$RemoveArgWithDupVar'(OrderPP(lp), vl);$
$RemoveArgWithDupVar'([], vl) = vl;$
$RemoveArgWithDupVar'(lp, []) = [];$
$RemoveArgWithDupVar'(p \triangleright lp, v \triangleright vl) =$
$\quad if(\pi_{\mathcal{V}}(p) \approx \pi_{\mathcal{V}}(v), RemoveArgWithDupVar'(lp, vl),$
$\quad\quad if(\pi_{\mathcal{V}}(p) > \pi_{\mathcal{V}}(v), v \triangleright RemoveArgWithDupVar'(p \triangleright lp, vl),$
$\quad\quad\quad RemoveArgWithDupVar'(lp, v \triangleright vl)));$

Afterwards we combine the constructed valuations using the function *MergeValuations*. If we find a transition relation, the valuation remains unchanged w.r.t. to the input valuation due to the conjunct $\pi_{\sigma'}(r) \approx s$. Hence we model deduction rules Def_1 as:

eqn $R_{Def_1}(p, s) = if(is_{def}(p), \{r{:}\mathcal{R}_{at} \mid at(\pi_{ac}(r), \pi_{p'}(r), \mathrm{S}) \in R(PES(\pi_{PE_{lab}}(p)), \mathrm{S})$
$\quad\quad \wedge \pi_{\sigma'}(r) \approx s \wedge\ is_{\surd}(\pi_{p'}(r))\}, \emptyset)$
whr $\mathrm{S} = MergeValuations(Assignments^{\sigma}(\pi_{ProcParAsss}(p), s),$
$\quad\quad RemoveArgWithDupVar(\pi_{ProcParAsss}(p), s))$
end ;

To model deduction rule Def_2 from Chapter 9.2.1 we have to specify (i) $\sigma[\overrightarrow{v'} \mapsto \{[\overrightarrow{d}]\}^{\sigma}]$ and (ii) $q[\overrightarrow{v} \mapsto \overrightarrow{v'}]$.

- To model (i) we specify $\{[\overrightarrow{d}]\}^{\sigma}$ with the help of function $Assignments^{\sigma}$. Then function *CreateVariableSubst* models $\overrightarrow{v'} \mapsto \{[\overrightarrow{d}]\}^{\sigma}$. Using the data expression $GenFreshVars(GetHighestId(s) + 1, GetVarLabelsFromPP(PE_{lab}(p)))$, we generate the fresh variables that are required for the assignments. The data expression $MergeValuations(\textsc{Ren}, s)$ merges two valuations such that it models $\sigma[\overrightarrow{v'} \mapsto \{[\overrightarrow{d}]\}^{\sigma}]$.

- To model (ii) we specify $q \in PE$ by $PES(\pi_{PE_{lab}}(p))$. Fresh variables are modeled by means of $GetVarLabelsFromPP(\pi_{ProcParAsss}(p)), GenFreshVars(GetHighestId(s) + 1, GetVarLabelsFromPP(\pi_{ProcParAsss}(p)))$. The fresh variables are subsequently used in *CreateVariableSubst* to model $\overrightarrow{v} \mapsto \overrightarrow{v'}$. The substitution is modeled by the already specified function *VariableSubstInProcessTerm*. The result is assigned to $\textsc{Subst}$.

By combining (i) and (ii) we model deduction rule Def_2 as:

$$
\begin{array}{lll}
\textbf{eqn} & R_{Def_2}(p,s) = \mathit{if}(\mathit{is}_{def}(p), \{r{:}\mathcal{R}_{at} \mid r \in R(\textsc{Subst}, \mathit{MergeValuations}(\textsc{Ren}, s)) \\
& \qquad\qquad \wedge \neg \mathit{is}_{\surd}(\pi_{p'}(r))\}, \emptyset) \\
& \textbf{whr} \quad \textsc{Ren} = \quad \mathit{VariableSubstInValuation}(\\
& \qquad\qquad \mathit{CreateVariableSubst}(\mathit{GetVarLabelsFromPP}(\pi_{ProcParAsss}(p)), \\
& \qquad\qquad\quad \mathit{GenFreshVars}(\mathit{GetHighestId}(s)+1, \\
& \qquad\qquad\qquad \mathit{GetVarLabelsFromPP}(PE_{lab}(p)))), \\
& \qquad\qquad \mathit{Assignments}^{\sigma}(\pi_{ProcParAsss}(p), s)), \\
& \qquad \textsc{Subst} = \mathit{VariableSubstInProcessTerm}(\\
& \qquad\qquad \mathit{CreateVariableSubst}(\mathit{GetVarLabelsFromPP}(\pi_{ProcParAsss}(p)), \\
& \qquad\qquad\quad \mathit{GenFreshVars}(\mathit{GetHighestId}(s)+1, \\
& \qquad\qquad\qquad \mathit{GetVarLabelsFromPP}(\pi_{ProcParAsss}(p)))), \\
& \qquad\qquad \mathit{PES}(\pi_{PE_{lab}}(p))) \\
& \textbf{end} \ ;
\end{array}
$$

9.4 Examples

The dogfooding approach captures the untimed semantics of the mCRL2 language in (roughly) 1000 lines of mCRL2 code. To validate that the approach can be used to study the semantics of a language, this section illustrates some of the models that have been analyzed. The implementation and all of the studied models can be found in Appendix B.5. The language specific parts are provided in Appendix B.5.1. The model specific parts are provided in Appendix B.5.2. The various input models are provided in Appendix B.5.3, where every initialization specifies a separate model.

Figure 9.1 illustrates six examples that were generated using the mCRL2 toolset (Release 2012, February). Every illustration corresponds to a generated LTS for a model in the meta notation, for which the mCRL2 representations are provided in the corresponding captions. An arrow depicts a transition. A node depicts a state. The initial state is indicated with a doubly lined node. A white colored node with outgoing transitions marks a non-terminating state, whereas a white colored state without outgoing transitions marks a terminating state. A gray colored state without any outgoing transitions marks a deadlock state.

The tools that have been used to generate the pictures are subsequently txt2lps, lps2lts and ltsgraph. The first tool reads a textual LPS and stores it in the binary LPS format. The second tool unfolds an LPS into an LTS. The third tool has been used to position the states of the LTS and to export the figures.

Figure 9.1(a) Figure 9.1(a) depicts the behavior for the mCRL2 process "$\tau + a_1 \cdot \delta$". The meta notated model that has been used corresponds to:

$$\mathit{Alt}(\mathit{Alpha}([\mathit{tau}]), \mathit{Seq}(\mathit{Alpha}([\mathit{Act}(a_1, [\,])]), \mathit{Deadlock}))$$

We assume an empty valuation, i.e., "$s = [\,]$".

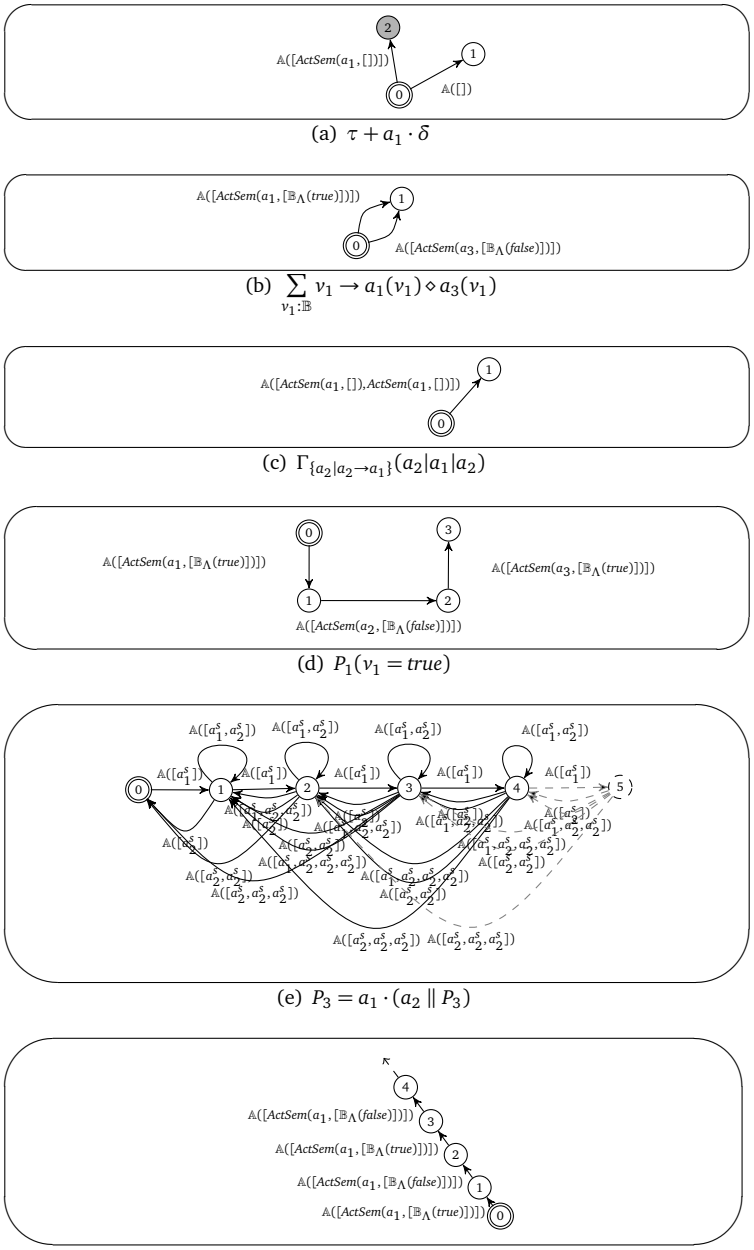


Figure 9.1 Six generated LTSs for different mCRL2 SOS input models

Figure 9.1(b) Figure 9.1(b) illustrates the behavior for the mCRL2 process "$\sum_{v_1:\mathbb{B}} v_1 \rightarrow a_1(v_1) \diamond a_3(v_1)$". The meta notated model that has been used corresponds to:

$$\begin{aligned}&Sum([\mathbb{B}_{\mathcal{V}}(v_1)], Cond_2(\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1)),\\ &\quad Alpha([Act(a_1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]), Alpha([Act(a_3, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]))))\end{aligned}$$

We assume that the initial data valuation is empty.

Figure 9.1(c) Figure 9.1(c) shows the LTS for the effect of a communication. We consider the native mCRL2 model "$\Gamma_{\{a_2|a_2 \rightarrow a_1\}}(a_2|a_1|a_2)$" that synchronizes the multi-action $a_2|a_2$ into the a_1 action. The meta notation that we use is :

$$Comm([communication([a_2, a_2], a_1)], Alpha([Act(a_2, []), Act(a_1, []), Act(a_2, [])]))$$

For generating the state space, we assume the initial data valuation to be empty.

Figure 9.1(d) The effect of local variables (i.e., the assignment of values to process parameters) is illustrated in Figure 9.1(d) for the native mCRL2 process:

proc $P_1(v_1{:}\mathbb{B}) = a_1(v_1) \cdot (P_2(v_1 = false) \cdot a_3(v_1));$
$P_2(v_1{:}\mathbb{B}) = a_2(v_1);$

The meta notation is defined as:

eqn $PES(P_1) = Seq(Alpha([Act(a_1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]),$
$Seq(Def(P_2, [ProcParAss(\mathbb{B}_{\mathcal{V}}(v_1), \mathcal{E}_{\Lambda}(\mathbb{B}_{\Lambda}(false)))]),$
$Alpha([Act(a_3, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]))));$
$PES(P_2) = Alpha([Act(a_2, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]);$

The initial model that is provided is "$P_1(v_1 = true)$" is in the meta notation reflected by "$P_1([argument(\mathbb{B}_{\mathcal{V}}(v_1), \mathbb{B}_{\Lambda}(true))])$". Initially we assume an empty data valuation. Observe the value changes of the Boolean variable v_1 in the data parameters of the actions a_1, a_2, and a_3.

Figure 9.1(e) Figure 9.1(e) shows the behavior for the recursive process definition of the native mCRL2 process "$P_3 = a_1 \cdot (a_2 \parallel P_3)$". The process allows more concurrent behavior every time the recursion is unfolded. For presentation purposes we have omitted the labels from the transitions. The corresponding meta model is:

eqn $PES(P_3) = Seq(Alpha([Act(a_1, [])]), Par(Alpha([Act(a_2, [])]), Def(P_3, [])));$

The initialization is provided through the mCRL2 process term "P_3" or, in the meta notation, "$Def(P_3, [])$". It is a valid mCRL2 specification but since the process introduces more concurrency at every recursion, it can not be linearized by the toolset. That is, to linearize an mCRL2 specification it must be either in the pCRL format [RGZW02]

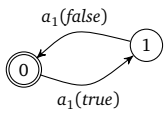


Figure 9.2 Generated LTS for the mCRL2 process $P_4(v_1{:}\mathbb{B}) = a_1(v_1) \cdot P_4(\neg v_1)$

or comply to the LPS format. Since the specification is none of the above, it cannot be used as input nor can we generate the corresponding state space. If we use the technique described in [SRW11b], we *can* generate the state space, because our framework produces models in the LPS format.

The model unfolds infinitely and we observe an exponential (unbounded) growth in computation time (and memory usage) to calculate the transitions. Therefore we only show the corresponding transitions for the first five states[‖].

Figure 9.1(f) Figure 9.1(f) shows the recursive mCRL2 process:

$$P_4(v_1{:}\mathbb{B}) = a_1(v_1) \cdot P_4(\neg v_1)$$

The process performs the action a_1 in which it shows the value of the Boolean variable v_1. After performing the action, we negate the value for variable v_1 and perform the process again. The state space for the native mCRL2 specification is depicted in Figure 9.2. We assume that v_1 is initially *true*.

Now, when we transform this process term into the meta notation we write:

eqn $PES(P_4) = Seq(Alpha([Act(a_1, [\mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))])]),$
$Def(P_4, [ProcParAss(\mathbb{B}_{\mathcal{V}}(v_1), \mathcal{E}^1_{expr}(\mathbb{B}_{\mathcal{O}}(neg), \mathcal{E}_{\mathcal{V}}(\mathbb{B}_{\mathcal{V}}(v_1))))]));$

For the state space of the meta notation model, we witness a non-terminating path (illustrated by a dotted line). If we compare the state spaces, we clearly see that the semantics between the two models deviates. The difference is caused by the generation and subsequently renaming of the fresh variables in a process definition in our approach, while the reuse of variables is enforced in the implementation. Investigation shows that the difference is caused by the generation and subsequently renaming of the fresh variables in a process definition. The mCRL2 semantics states that every unfold of a process definition introduces a (fresh) variable. This means that we will never visit a previous visited state for which the process term and the data valuation are identical.

[‖]For illustration purposes we added a 6th state, and renamed the transition labels $\mathbb{A}([ActSem(a_1, [])])$ and $\mathbb{A}([ActSem(a_2, [])])$ to a_1^s and a_2^s respectively.

Missing operator	Added missing operator
$(Allow_2)\dfrac{\begin{array}{c}\underline{m} \cap (V \cup \{\tau\}) \neq \emptyset, \\ (p,\sigma) \xrightarrow{m}_t (p',\sigma')\end{array}}{(\nabla_V(p),\sigma) \xrightarrow{m}_t (p',\sigma')}$	$(Allow_2)\dfrac{\begin{array}{c}\underline{m} \cap (V \cup \{\tau\}) \neq \emptyset, \\ (p,\sigma) \xrightarrow{m}_t (p',\sigma')\end{array}}{(\nabla_V(p),\sigma) \xrightarrow{m}_t (\nabla_V(p'),\sigma')}$

Table 9.1 Example of a missing operator in a deduction rule

9.5 Discovered Issues

Dogfooding the mCRL2 language forced us to closely look at the defined formal semantics. Although the language is formal, the semantic definitions still contained ambiguous behavior. To illustrate, the original specification states: "let there be a fresh variable d'". Does it mean that d' is a unique fresh variable or do infinitely many fresh variables correspond to d'? We assumed the first since it defines behavior that can be (exhaustively) simulated. The second option would correspond to infinite branching behavior for every freshly introduced variable.

Dogfooding also led to the definition of a semantic multi-action equivalence class. The original semantics specifies a multi-action as a collection of semantic actions. It assumes that the semantic multi-action is an equivalence class, however this is never explicitly stated, but is required for the defined functions on the inference rules. The statement has been made explicit in Chapter 2.2.

Dogfooding the formal semantics revealed semantic issues and implementation deviations. Although the semantics has been considered finalized since September 2009, we still discovered errors. These errors include simple oversights in the documentation such as duplicate deduction rules (e.g., for $\|$) and a missing deduction rule for the parallel operator. We indicated that seven auxiliary operators were missing from the deduction rules. One of them is illustrated in Table 9.1.

Furthermore, we uncovered two deviations between the semantics and its implementation. The first deviation has been discussed in Chapter 9.4, where an iteration introduces infinite behavior. The second semantic deviation is illustrated by the following example and the original definition for deduction rule:

$$(Sum_2)\frac{\begin{array}{c} w \in M_D, \\ (p,\sigma[v \mapsto w]) \xrightarrow{m}_t (p',\sigma') \end{array}}{(\sum_{v:D} p,\sigma) \xrightarrow{m}_t (p',\sigma')}$$

Under this assumption, we consider the following mCRL2 process:

proc $\quad P = \sum_{d:\mathbb{B}} a(d) \cdot b(d);$

The process selects a Boolean value and assigns it to the variable d, and performs the actions $a(d)$ followed by $b(d)$. We define process Q as:

proc $Q = a(true) \cdot b(true)$
$\quad + a(false) \cdot b(false);$

Note that P and Q are strongly bisimilar, thus $P \parallel P \leftrightarrows Q \parallel Q$. If we specify the models in both the native representation and the meta notation, and generate their state spaces, the state spaces must be (strongly) bisimilar. However, when we performed the bisimulation check** we observed that the state spaces where *not* strongly bisimilar. The counter example showed that $P \parallel P$ could perform the actions $a(false) \cdot a(true) \cdot b(true) \cdot b(true)$, whereas $Q \parallel Q$ did not. The cause was related to the assignment of values to binder variables. If a value was already assigned to a binder variable, the selected variable was overwritten in the valuation. Because of the interleaving behavior of P and the sum operator usage, it resulted in undesirable behavior. To repair the undesirable behavior, we redefined deduction rule Sum_2.

Chapter 8.3 proves the isomorphic relationship between the formally engineered models and the behavior described by the native models for rules in the De Simone-format. When we assume that other SOS formats preserve the same relationship, we see that the proven relationship is stronger than the one that can be guaranteed by the LPS-tools in the mCRL2 toolset. These tools (including the linearization of an mCRL2 specification) only guarantee the behavioral equivalence for strong bisimulation (unless stated otherwise). Hence, native mCRL2 models may depict different state spaces than models that have been generated with the semantic engineering approach.

9.6 Implementation

The implementation of the semantic engineering approach creates for every deduction rule a separate data equation. Thus, if a formal language has many deduction rules, the transformation results in a (large) number of data equations. Because the approach is applied straightforwardly, the resulting models are not optimized. Hence, we were not yet able to explore the state spaces for models like the ABP [BSW69].

The limited model exploration can be devoted to several reasons. Firstly, many data equations specify duplicate rewrite steps. Rewriting is performed without any caching. Hence, the same rules are rewritten multiple times for the same input. Take the mCRL2's parallel operator for example: $Par_1, \ldots, Par_8$ in Table 2.4. The premises of the eight deduction rules share computations that are individually rewritten. The performance could be improved by rearranging the computations, however at the loss of the readability and traceability w.r.t. the deduction rules.

Another reason concerns the specification of the deduction rules. To compute the set of transition relations, all information should be available prior to computing the transitions. Any modification that needs to be applied on a transition, after it has been computed is expensive. To illustrate, we refer to deduction rule Par_8 in Table 2.4. Although it seems as a trivial task to resolve and substitute double occurring variables, it is a rather expensive task, because we first compute the transition relations and then

**tool: *ltsconvert -ebsim*

resolve the valuation, for which many (needless) rewriting steps are performed. To illustrate, we conducted a couple of tests in which we removed certain deduction rules. By only removing deduction rule Par_8, the state space generation increased to 700 states per minute rather than 50 states seen prior.

The third reason concerns the implementation of the rewriter. The rewriter operates on abstract terms that represent all functions, variables and values. This applies to the complex data structures, but also to Booleans and numbers[††]. Therefore whenever an operation is performed on such a basic sort, it is handled by the mCRL2 rewrite engine, and not directly by the (optimized) machine specific instructions. Since the computations are executed on a higher level, performance is lost.

9.7 Related Work

Dogfooding is applied in (ordinary) software development for developing new, or extending existing (software) products. Examples are found in the argument of compilers [Ter97], where bootstrapping is applied in compiler construction. Other examples include the Eclipse framework [Haa] that develops plug-ins for the extension of the Eclipse framework. Another considered example is the use of editors like Emacs/Vi. Here, the editors are used to write customizations for the editors themselves. Wolfram Research states [McL07] that (parts) of their web sites, applications, documentations, and test and build processes are driven by the *Mathematica* Language.

Practicing these techniques in formal software engineering, especially in the area of formal languages and model checkers, is uncommon. Especially, a model checker that eats and interprets the formal semantics of its own language. We believe that our dogfooding approach is unique and, in that sense, the first of its kind. However, it still leaves the question, are their alternative approaches? To the best of our knowledge, Maude [EM02] would be the only candidate that could directly express the SOS of the mCRL2 language. Maude is a high-level language and high-performance system supporting both equational and rewriting logic and is used for, and applied to, a wide range of applications. Its simple and expressive logic allows the representation of many models of concurrent and distributed systems, including forms of SOS.

To illustrate, we briefly list some of the work that has been carried out using Maude. In [BHMM02] the authors translate Modular SOS (MSOS) [Mos04a, Mos04b] to the Maude rewriting logic and prove the transformation correct. In [BV07] the authors model GSOS/OSOS rules in the Maude system, allowing them to execute Ordered-SOS specifications. The work of [HHVOM07] implements Eden (the parallel extension of the functional language Haskell [HHJW07]) in Maude. More recent work [RRH10] implements the semantics for the πCRWL calculus and the formalization of AADL in [ÖBM10]. Based on these, and other successful experiments, we believe that the semantics of the mCRL2 language can be implemented using Maude. As this route is still open, we consider the alternative implementation to be future work.

[††]The mCRL2 language uses mathematical numbers which are not restricted by any machine constant.

9.8 Conclusions

This chapter shows the process of dogfooding the mCRL2 language to its own toolset, despite: "Engineers who use their own company's tools exclusively, tend to propagate the bad aspects of their tools because they might not even realize an alternative approach exists. They often fail to either understand or appreciate the good points of other companies' tools. Furthermore, it also encourages the *Not Invented Here syndrome* [Har06]".

The semantic approach illustrates that the mCRL2 formalism can specify *and* simulate the Structural Operational Semantics (with a given model) of its own language. To succeed, the TSS needs to be an *mCRL2-restrictive* TSS (Definition 8.2.1), and the deduction rules (along with auxiliary and supporting functions) are captured by sorts and data equations. For the computational feasibility, it is required that the data equations specify a finite rewrite sequence. Moreover, the enumeration over dense domains and functions need to be avoided. The transformation of the language concepts and their formal semantics are a non-trivial task. Hence we outline and motivate the underlying design and modeling decisions. The application of the approach is illustrated by examples, their generated state spaces, and discovered mismatches in the definition, implementation and specification of the mCRL2 language.

The semantics of the mCRL2 language is rather rich. Hence, we are confident that other (formal) languages, such as CSP [Hoa78], CIF [BRSR07], ASML [Bör98] and POOSL [vdPV97] can also be subjected to the approach. The framework can also be used to formalize and validate the behavior for various domain specific languages.

The integration into language workbenches is considered as a future activity. A language workbench could support the definition of signatures and the associated SOS and include an automated transformation from deduction rules to data equations. The latter nearly constitutes an one-to-one mapping when considering the language specific parts of our mCRL2 implementation. Such a tool is useful, since a manual implementation is tedious, time consuming and prone to errors. The conversion from the syntactic instance of a model to its syntactic meta notation could be automated. Because the semantic interpretation is not bound to a single language, it theoretically allows the study of compound concepts from different formal languages within a single mCRL2 specification. Another direction could include the heterogeneous composition of native mCRL2 models and semantic models from other languages.

This work can be extended by including the timed fragment of the mCRL2 language. As a direct interpretation of the dense time domain would pose all kinds of problems, it might be worthwhile to consider an approach that partitions the dense domains into a discrete/non-dense domains. Considering these partitioning rules could be included as a part of the formal semantic definition.

In conclusion, we like to emphasize that our approach can be applied to, and implemented in, other languages and toolsets, if they support the definition and computation of set comprehensions, deal with quantifiers, and support a mechanism to systematically perform transitions (i.e., they model an LPS).

Chapter 10

A Reflection on the Semantic Bridge

10.1 Introduction

The second part of the thesis describes a method to semantically engineer models suitable for model checking by systematically transforming the semantics of a formal language. The first chapter of the second part illustrates the formalization of an informal language. The second chapter describes the kernel of the semantical engineering approach. The third chapter demonstrates the capability of the approach by transforming the semantics for a formal general purpose language.

In essence, the approach encompasses a transformation that takes a Transition System Specification (TSS) and produces an LPS. The data specification of the LPS describes the transitions that are valid for the deduction rules. When a model is provided, transitions can be computed using the data specification and the rewrite engine that supports the mCRL2 toolset. The LPE within the LPS serves as a transition generator for the computed transition relations.

This chapter elaborates on the merits and the encountered restrictions while applying the semantic bridge. We elaborate on the correspondence relation between the behavior defined by the formal abstract models and the exhibited behavior during the analysis in Chapter 10.2. The incorporated restrictions are discussed in Chapter 10.3. The suitability of the approach is described in Chapter 10.4.

10.2 Model Correspondence

The behavior described by the LTS of an original model and a semantically engineered model, for which the semantics complies to the De Simone format, are isomorphic

(Chapter 8.3). Chapter 9 shows that we can model and simulate rule formats that are more complex. Hence, we believe that many rule formats can be modeled in a similar fashion. When the modeled deduction rules can be expressed in the mCRL2 syntax, and they can be computed by the underlying rewrite mechanism, we believe that it is possible to generate an LTS that captures the execution behavior of a model. We also believe that the behavior expressed by models of other format rules, and the behavior expressed by their semantically engineered models, are isomorphic as well. More (empirical) research is required to support these presumptions.

The semantical engineering approach establishes a close relation with native models in the (abstract) syntax. Although it appears that the state information is invisible to an observer, it can be made public by adding additional (self loop) transitions: the LPS stores the state information in its process parameters. State information can be made visible in a similar way as we have seen with predicates (Chapter 8.6). Exposing the state of the model can provide design engineers additional and helpful information about the conducted analysis (e.g., the state in which a system deadlocks).

As we already explained in Chapter 6, it is difficult to relate the results of an analysis from hand crafted verification models back to the models used by engineers. Especially, when ad-hoc abstraction techniques are applied. As the semantical engineering approach is (i) free from any abstraction techniques and (ii) directly operates on the abstract language, engineers should be able to understand the models that are used during the analysis.

In Figure 10.1, we illustrate the relationship between the different (meta)-models. The concrete syntax models at the top-left are created by the design engineer. The abstract syntax models at the top-middle are derived from the concrete models. To ensure that the models are syntactically sound, they must correspond to the syntax's signature that is explicitly defined by the TSS. The abstract models that serve as input for the formal analysis are transformed into one or more process parameters of the LPS. The transformation of the SOS deduction rules into data equations is denoted at the bottom of the figure. The processable semantical engineering route is represented by the solid black arrows. The inverse way of the engineering route, the route that relates the analysis model to the original model, is represented by the dotted arrows. Based on the conducted case studies, we believe that the route from the original model to the analysis model can be automated. Although we have no empirical evidence, we believe that it can also be done for the inverse route.

10.3 Restrictions

The execution of the semantical engineering approach heavily depends on the underlying rewriting technology of the mCRL2 toolset. Therefore the modeling has to be done in a particular style to ensure computational feasibility. To ensure that the models can be analyzed and/or executed, we here state the encountered restrictions:

1. Specifications must be *mCRL2-restrictive* TSS (Definition 8.2.1).

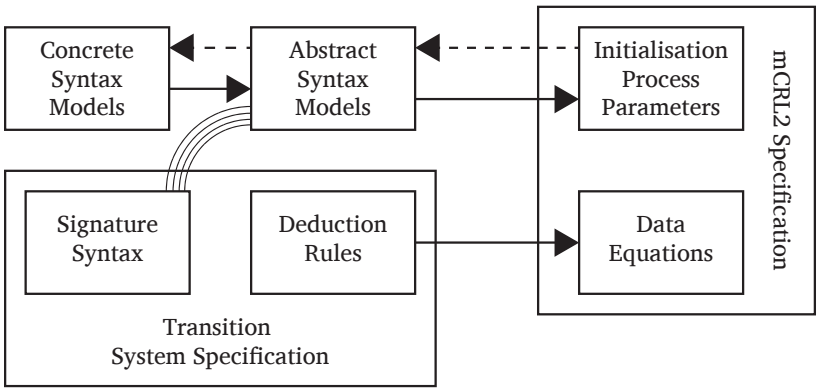


Figure 10.1 Relating the (meta)-models in the semantical engineering approach

2. Deduction rules (along with auxiliary and supporting functions) must be expressible by sorts and data equations.

3. Rewrite rules must be terminating.

4. Enumerations over dense domains (e.g., $\mathbb{R}$) and functions need to be avoided.

To fulfill item 1, the signature of a process term must have a finite set of symbols and a finite set of action labels. If we use a language that does not comply to this requirement, it can become impossible for the underlying rewriter to enumerate and find all valuations that satisfy a solution. Moreover, the TSS must specify a finite set of deduction rules and must have a (strict) stratification [MRG07], e.g., when dealing with negative premises. Since every deduction rule corresponds to a single data equation, and the set of mCRL2 data equations is required to be finite, the set of deduction rules must be finite as well.

For item 2, we express all the concepts of a formal language in sorts, data expressions and data equations that meet the syntactic requirements of the mCRL2 language. This means that some of the widely accepted mathematical syntax is molded into mCRL2 notations. Examples can be found in the notation of set comprehensions, the specification of tuples, or the way in which term substitutions are performed. Hence, some notations may deviate from typical mathematical notations.

To comply to item 3 all rewrite rules on (open) terms require a rewrite strategy that rewrites an expression in a finite number of steps. This means that any recursive (non-guarded) rewrite rule can pose potential problems. This especially holds for the rewrite rules that are used to compute set comprehensions. To illustrate the problem, assume that function $g{:}A \to 2^A$ defines a set comprehension, $f{:}A \to \mathbb{B}$ is a Boolean

function, $h{:}A \rightarrow A$ denotes a function on A and we define g as:

$$g(p) = \{a{:}A \mid f(p) \wedge \ a \in g(h(p))\}$$

Since the current rewrite strategies in the mCRL2 toolset assume no order, it is possible that $a \in g(h(p))$ is rewritten prior to $f(p)$, which results in an infinite sequence of rewriting steps. Hence, we recommend that the computation of finite functions is performed prior to the (possible) recursive ones. Provided that f can be computed independently from the body of the set comprehension, we introduce a guarding *if* construction for which f must hold prior to rewriting the body. So we alternatively write:

$$g(p) = \mathit{if}(f(p), \{a{:}A \mid a \in g(h(p))\}, \emptyset)$$

The rewrite rules specify that the condition from *if* is rewritten prior to its branches. Hence, we force a rewrite sequence that is fixed. This technique has been applied in Chapter 9 to determine whether a term is of a certain form.

Restriction item 4 ensures that the resulting mCRL2 models are analyzable. As already pointed out in the previous section, enumeration over dense domains between any two different values, results into an uncountable number of solutions. Also the enumeration over functions is not supported. Therefore we advise to only use dense domains and functions if the enumeration over them can be avoided. If either one is used, without any post-processing, it renders an (exhaustive) analysis impossible.

10.4 Suitability

An (informal) DSL's syntax is defined through some grammar or meta-model. The semantics of a DSL is however implicitly defined in a translation to another (informal) language or implemented in some execution engine or interpreter. Hence, we exhibit unexpected behavior during the execution of these models. To combat these abnormalities, we could formalize the language. However for an industrial DSL this is a challenging task. Particularly when considering the operational impact of changing the existing semantics. If one formalizes a language, as shown in Chapter 7, and applies the engineering route, from Chapter 8, an engineer can study the language and reason on the executed behavior. So if one defines the DSL's semantics through SOS, we are able to aggregate and compose terms and study the behavior of composed terms in isolation. The result is a language definition where the DSL's abstract and concrete syntax are formally related to its static and dynamic semantics. Our approach provides a handle to analyze different aspects, e.g., throughput and safety, that are closely related to the executed behavior. This also facilitates means, to study the impact of changing the semantics for particular language constructs. The use of separate operators can limit the regression and qualification impact in an operational context. Recall that in Chapter 7, we have added complementary operators to disambiguate the language. The proposition of these operators, along with an explanation on the disambiguated behavior, was supported by the semantical engineering approach from Chapter 8.

10.4.1 Language Prototyping

The approach is suitable for engineering or prototyping a behavioral DSL or a formal language. As demonstrated in previous chapters, the semantic bridge is not bound to a particular language. In fact, for any behavioral language (for which the behavior can be expressed in a TSS under the modeling restrictions), we argue that it is possible to define a semantic bridge, transform the models and analyze its behavior.

Language prototyping requires a high degree of flexibility, because language concepts can rapidly change. As the changes are modeled with relatively low effort by an engineer, the associated behavior can easily be studied and explored. In turn, this would reveal mistakes between the engineer's intended semantics and the DSL's defined semantics more easily, since the approach facilitates an automated analysis for concrete models.

When a language stabilizes one can choose to implement the deduction rules into code, instead of modeling it in a modeling environment. If one implements the deduction rules (directly) into code, it facilitates the execution of a model on a dedicated system or an architectural platform. If one chooses to provide a native implementation, one could use the semantic bridge as a safeguard, to determine whether the executed transitions by the implementation also appear as transitions in the model.

10.4.2 Integration into Development

The work of [Del11] has been performed to test the suitability of the semantical engineering approach within an industrial setting. Here, model driven engineering techniques are used to support the evolution and maintenance of a DSL. Build on top of the starting point of the semantical engineering approach i.e., the formal abstract syntax and the transformed set of deduction rules, the author transforms the concrete syntax into the required formal abstract syntax, by means of a model-to-model transformation in a prototype model driven engineering environment.

By relatively little effort, the author integrates the semantic bridge into an actual development trajectory. Although we deal with a prototype, we believe that the approach can be made more mature such that it can be incorporated into a wider set of development trajectories. By establishing such an integration, design engineers could fabricate their own formal languages more easily.

10.4.3 Separation of Concerns

The semantical engineering approach provides a separation of concerns for the design, the use and the analysis of a formal language. Here we identify three concerns.

Firstly, the semantics and syntax are separated. Although syntax and semantics are tightly related, they can be developed independently. That is, a language engineer can develop or change the semantics for a piece of syntax compositionally and independently of other language constructs, in a clear and a precise manner.

Secondly, when all syntactic elements have formal semantics, and the elements can

be composed compositionally, it offers design engineers the possibility to experiment and independently study the model's behavior. While a design engineer can freely compose terms, a language engineer can regulate the notions that are made public.

Thirdly, the analysis engineer can focus on the verification of properties. This task can be carried out independently from other engineering disciplines. The engineer responsible for the verification can optimize the resulting models and apply the required manipulations and/or abstractions such that properties are successfully verified.

10.4.4 Maintainability

To facilitate release management, companies typically have some sort of version control to separate releases and development branches. During development interfaces, systems, components and modules change over time. When parts are not under version control, it results in an additional integration effort. Therefore the amount of resources that are required to preserve a mapping, indicate a maintenance degree. Since maintenance is equally important as development, it should not be overlooked.

Because the design of a language, the design of models and the transformation from a specification environment to an analysis environment can be automated, we believe that the semantic bridge can be incorporated into a release management system. Simply because no more ad-hoc transformations are applied, changes to the individual components (i.e., the language, the model and the transformation) can all be kept local. The implementation of the changes can be made processable. Hence, the semantic bridge can be perceived as processable semantic engineering route. If we compare the maintainability effort for the semantic approach to the traditional or ad-hoc bridges (as discussed in the first part of the thesis), we argue that the semantic approach requires fewer resources when fully matured.

10.4.5 Reusability

It is not unthinkable that different languages cover the same semantic definitions by different abstract notations or that abstract notions with the corresponding semantics are identical. Since SOS' deduction rules are defined in a compositional and independent manner, and the semantic bridge treats them similarly, we believe that the semantic bridge can be used within a component-based framework for the design, the specification and the implementation of programming and behavioral domain specific languages. This means that if notions are shared between different languages, it should be possible to reuse those parts for which the syntax and semantic specifications are identical. A similar point of view is reflected in [dBIM06].

Moreover, as different languages are typically developed independently by different people at different times, the deduction rules can act as a platform, where different engineers can study and reuse the work that has been performed by others. As a result, it can lower the technological diversity during the realization of a new language. As the diversity can be kept to a minimum, it should have a positive impact on the reusability and maintainability of languages.

Chapter 11

Conclusions

This thesis presents several techniques for engineering formal behavioral models. Formal behavioral models are important, because they are used to unambiguously study the behavior of a system, piece of software or protocol. In conjunction with modal properties, these models can be subjected to formal verification that determines whether the models possess the desired properties. Unfortunately, many formalisms are informally defined or provide no suitable techniques to formally verify models. Hence, it is important that these models can be transformed to models that facilitate these analysis. Thereto we have surveyed the following two research questions:

I *"What are practiced methods to create formal behavioral models, suitable for verification?"*, and

II *"Can we systematically administer formal techniques to translate behavioral models into a formalism that facilitates formal verification?"*

11.1 Contributions

The first research question is investigated by Part I of the thesis. Part I illustrates several ad-hoc modeling techniques that are commonly practiced to derive formal models from (in)formal specifications. Models that are constructed using these ad-hoc modeling techniques belong to the class of *syntactically engineered models*.

Chapter 3 presents a comparison case study that examines the expressiveness of different languages. Here the mCRL2 specification language is compared to the languages TLA+ [Lam02], Bluespec [HA00], Statecharts [Har87], and ACP [BK84]. The comparison is carried out for the models that are constructed for the 2×2 switch buffer [Blu05], according to a set of system descriptions. The case study shows that the mCRL2 language is suitable to model these descriptions. In addition, the case

study also shows that the constructed models can be verified with the help of the accompanied mCRL2 toolset.

Chapter 4 illustrates the transformation from a software implementation to a formal model. The software originates from a system used to print Printed Circuit Boards. The controller is implemented by a third party. The formal model is subsequently verified for having various safety properties. Because software implementations are in general too complex to be analyzed directly, the method abstracts from the values of all program variables. The abstraction results in an over-approximation with respect to the original behavior, for which only the interface calls between the processes and the non-deterministic choices in the bodies of the procedures remain. The models are specified in the mCRL2 language and model checked with its toolset.

Chapter 5 demonstrates the denotational transformation between two formal languages. Here, the Chi 2.0 language is translated into the mCRL2 language. The transformation is defined because the toolset that is associated with the Chi 2.0 language is only suitable for the simulation of hybrid systems and provides no native means for conducting verification. The non-trivial transformation scheme translates syntactic notions into correlating mCRL2 notations.

Chapter 6 explains how verification results can be visually disseminated to various disciplines. The solution integrates formal methods with the physical designs from industrial systems. By combining the physical designs with a trace from the formal model, we are able to animate the behavior of a system. The dissemination is accompanied with an exploratory industrial case-study that models the behavior of a wafer dryer facility. The behavior has been modeled in the mCRL2 language and subsequently been model checked for deadlocks. Since the counter examples are difficult to understand by other disciplines, they are animated through the physical designs of the dryer facility.

The aforementioned model transformations are all performed ad-hoc and require some form of human ingenuity in order to succeed. Simultaneously, the same human ingenuity can potentially introduce undesired or unintended behavior which may stay unnoticed.

The observations made in the ad-hoc methods, pose the second research question. This question is answered in Part II by introducing a model-to-model transformation that is more rigorous. The technique that is proposed and demonstrated takes the formal semantics of a language, converts the semantics to a set of functions and executes models with the help of these functions. Models that are constructed in this way belong to the class of *semantically engineered models*.

Many behavioral specification languages are informally defined. Frequently, the behavior is captured by some informal description or specified through some interpreter. If one wants to semantically engineer models, on needs to formalize a language first. Hence, Chapter 7 presents the formalization of an industrial domain specific language, called TRECS [Nie04]. The result of the formalization is captured by a Transition System Specification (TSS) [Gro93, BG96], for which the (dynamic) semantics are expressed by Structural Operational Semantics (SOS) deduction rules. By formalizing the language in a compositional way, we show that it is possible to

assign a formal semantic description, even for a language that is used in an industrial setting. During the formalization we discovered several ambiguities, that have been resolved by taking informed choices.

Chapter 8 describes the kernel of the approach to semantically engineer models. Here we take a TSS and transform it into a restricted mCRL2 specification, namely a Linear Process Specification (LPS). The signature of a process term is transformed into a structured sort, the deduction rules are modeled as a set of data equations, and the transition relations are modeled by different actions. The transformation is explained for deduction rules that are in De Simone rule format. The resulting mCRL2 specifications can be directly analyzed with the mCRL2 toolset.

Chapter 9 demonstrates the applicability of the approach by taking the TSS that belongs to the untimed fragment of the mCRL2 language. Since the behavior is manually implemented in the underlying toolset, it is impossible to guarantee that the implemented semantics corresponds to the specified operational semantics. To validate that the semantics corresponds, we directly take the SOS deduction rules and transform them into mCRL2 data equations. Thus we basically feed the mCRL2 toolset its own formal language definition. We elaborate on the underlying design decisions for modeling the syntax and semantics into an mCRL2 specification, describe the transformation of the deduction rules, and illustrate the experiments that have been conducted. Despite its formal characterization, thorough study and broad use in many areas, the semantic dogfooding approach revealed a number of (subtle) differences between the mCRL2's intended semantics, the defined semantics and implemented semantics.

Chapter 10 reflects on the method that semantically engineers models. We state the potential benefits and the implied restrictions.

All of the work that is described in this thesis has been performed as work for the *TWINS: Optimizing Software Hardware Co-design Flow for Software Intensive Systems*" project or has been carried out in the "Kenniswerkingsregeling" project "*LithoSysSL*". The results are presented in a number of (scientific) publications:

> [GKM+08, SSR08a, SSR08b, MSW09, SR09, SRG09, SSR09, SRG10, GKS+11, SRGW11, SRW11a, SRW11b, SWR+11a, SWR+11b, SRWG12]

11.2 Future Work

The technique to *syntactically engineer models* is often used to create formal models for feasibility studies, sanity checks on designs, reverse model engineering, and ad-hoc modeling approaches in general. Although the technique is practiced in industry, it often does not seamlessly connect to the development process; it requires ingenuity. The technique to *semantically engineer models* requires less ingenuity during the transformation, but more language engineering effort. When applied, it offers a more processable way of working and better integration with existing industrial designs for specifying behavior. Although the semantical engineering approach looks promising, *scalability* and *computability* still require investigation.

The semantic engineering approach focuses on the application of the method, rather

than the optimizations that can (and should) be explored to verify large industrial systems. In all of the shown examples, we directly transform the deduction rules into corresponding data equations. Because the underlying rewriter of the mCRL2 toolset does not facilitate any form of caching, every deduction rule is completely and separately computed, even when the same sequence of rewriting steps can be shared. In some cases this may result in an (hyper)exponential growth on the number of rewriting steps that are (supervacaneously) performed. To eliminate these hurdles, we need to develop (founded) techniques that increase the scalability by applying different rewrite strategies, rearrange/merge computations, and/or use axioms to normalize terms.

Finally, we want to consider the computability of deduction rules. As deduction rules can practically specify all forms of mathematics (e.g., even undecidable clauses), boundaries must be investigated, defined and set. This may include research in the field of (higher-order) rewrite systems, deduction rules that can be translated and accepted by rewrite systems, the applicability of the approach by other formalisms, or semantic extensions that incorporate time, probabilistics, and continuous behavior.

Appendix A

Proofs

A.1 Correspondence Relation Between Chi 2.0 and mCRL2 Specifications

The denotational translation from Chapter 5 transforms a Chi 2.0 model into an mCRL2 model. The correspondence relation between the different models is explained here. We provide the relation through the structure of the LTSs, modulo the order of the actions in the multi-action.

Let $\oplus : Time_{\mathrm{H}} \times \mathbb{R} \rightarrow Time_{\mathrm{H}}$, be:

$$h \oplus d \stackrel{\triangle}{=} if(d \approx 0, time_{\mathrm{H}}(\pi_{time}(h), \pi_{counter}(h)+1), time_{\mathrm{H}}(\pi_{time}(h+d), 0))$$

Then the transition relations between the formalisms is explained in Appendix A.1.1 and Appendix A.1.2.

Remark A.1.1. The correspondence relation is provided without proof. The translation is provided to demonstrates that a compositional transformation for a subset of the Chi 2.0 language is feasible. Providing the proof that the above relation holds is considered as future work.

A.1.1 Relating Chi 2.0 to mCRL2

Assume that $\mathcal{F}_{\mathrm{init}}\left(\langle p, \sigma, E\rangle\right) = X$ and $\mathcal{F}_{\mathrm{init}}\left(\langle p', \sigma', E\rangle\right) = X'$.

Action transitions

Actions: Chi 2.0 action transitions that are labeled with an action l which originates from the set of basic action labels, i.e., $l \in \mathcal{L}_{\mathrm{basic}}$, are related to mCRL2 action

transitions in the following way:

$$
\begin{array}{l}
E \Vdash \langle p, \sigma \rangle \xrightarrow{\sigma, l, W, \sigma'} \langle p', \sigma' \rangle \Rightarrow \\
\quad \exists_{c,c':\mathbb{N}} \\
\quad X \xrightarrow[]{\left(\begin{array}{c} com_{\text{time}}(\sigma(\mathsf{time}),c)|\Big|_{v \in D_{\mathbf{disc}}} com^{[\![v]\!]}_{\text{mem}}(\sigma(\vec{v}))|l| \\ diff(W)|\Big|_{v \in dom(\sigma')} com^{[\![v]\!]}_{\text{mem}'}(\sigma'(v))|com_{\text{time}'}(\sigma'(\mathsf{time}),c') \end{array}\right)}_{time_{\text{H}}(\sigma'(\mathsf{time}),c')} X'
\end{array}
$$

Communicating actions: Chi 2.0 action transitions that are labeled with an action h which originates from the set of successful communication actions, i.e., $h \in \mathcal{H}$, relate to mCRL2 action transitions in the following way:

$$
\begin{array}{l}
E \Vdash \langle p, \sigma \rangle \xrightarrow{\sigma, h!?x, W, \sigma'} \langle p', \sigma' \rangle \Rightarrow \\
\quad \exists_{c,c':\mathbb{N}} \exists_{W', W'' \subseteq \mathcal{V}} \\
\quad X \xrightarrow[]{\left(\begin{array}{c} com_{\text{time}}(\sigma(\mathsf{time}),c)|\Big|_{v \in D_{\mathbf{disc}}} com^{[\![v]\!]}_{\text{mem}}(\sigma(\vec{v}))|comm_{[\![h]\!]}(x)| \\ diff(W')|diff(W'')|\Big|_{v \in dom(\sigma')} com^{[\![v]\!]}_{\text{mem}'}(\sigma'(v))|com_{\text{time}'}(\sigma'(\mathsf{time}),c') \end{array}\right)}_{time_{\text{H}}(\sigma'(\mathsf{time}),c')} X' \\
\quad \text{where } W = W' \cup W''
\end{array}
$$

Internal actions: Chi 2.0 action transitions that are labeled with an internal action τ, which do not originate from an hidden communication, are related to mCRL2 action transitions in the following way:

$$
\begin{array}{l}
E \Vdash \langle p, \sigma \rangle \xrightarrow{\sigma, \tau, W, \sigma'} \langle p', \sigma' \rangle \Rightarrow \\
\exists_{c,c':\mathbb{N}} \\
X \xrightarrow[]{\left(\begin{array}{c} com_{\text{time}}(\sigma(\mathsf{time}),c)|\Big|_{v \in D_{\mathbf{disc}}} com^{[\![v]\!]}_{\text{mem}}(\sigma(\vec{v}))| \\ diff(W)|\Big|_{v \in dom(\sigma')} com^{[\![v]\!]}_{\text{mem}'}(\sigma'(v))|com_{\text{time}'}(\sigma'(\mathsf{time}),c') \end{array}\right)}_{time_{\text{H}}(\sigma'(\mathsf{time}),c')} X'
\end{array}
$$

Chi 2.0 action transitions that are labeled with an internal action τ, which originating from an hidden communication, are related to mCRL2 action transitions in the following way:

$$
\begin{array}{l}
E \Vdash \langle p, \sigma \rangle \xrightarrow{\sigma, \tau, W, \sigma'} \langle p', \sigma' \rangle \Rightarrow \\
\quad \exists_{n:\mathbb{N}} \exists_{W', W'' \subseteq \mathcal{V}} \\
\quad X \xrightarrow[]{\left(\begin{array}{c} com_{\text{time}}(\sigma(\mathsf{time}),c)|\Big|_{v \in D_{\mathbf{disc}}} com^{[\![v]\!]}_{\text{mem}}(\sigma(\vec{v}))|diff(W')|diff(W'')| \\ \Big|_{v \in dom(\sigma')} com^{[\![v]\!]}_{\text{mem}}(\sigma'(\vec{v}))|com_{\text{time}'}(\sigma'(\mathsf{time}),c') \end{array}\right)}_{time_{\text{H}}(\sigma'(\mathsf{time}),c')} X' \\
\quad \text{where } W = W' \cup W''
\end{array}
$$

Continuous behavior

The continuous behavior for a Chi 2.0 specification, i.e., the progression of time, is related to an mCRL2 idle transition in the following way:

$$E \Vdash \langle p, \sigma \rangle \overset{t,\sigma,(\theta_n,\theta_s,\theta_r)}{\longmapsto} \langle p', \sigma' \rangle \Rightarrow \exists_{c:\mathbb{N}}\, X \rightsquigarrow_{time_{\mathrm{H}}(\sigma(\mathsf{time}),c)\oplus t}$$

A.1.2 Relating mCRL2 to Chi 2.0

To relate the behavior of an mCRL2 specification to a Chi 2.0 specification, we assume that if we perform an mCRL2 transition from a state, it relates to a state in a Chi 2.0 specification. We assume that every mCRL2 p is related to Chi 2.0 model p_χ via function $\mathcal{F}^{-1}$, i.e., $\forall_p \exists_{p_\chi} \mathcal{F}^{-1}(p) = p_\chi$. Furthermore, we assume that for the modeled discrete model variables $\mathcal{V}_{\mathrm{mCRL2}}$ the condition $\mathcal{V}_{\mathrm{mCRL2}} \subseteq dom(\sigma)$ holds.

Action transitions

Actions: An mCRL2 action transition that models the behavior of an $l \in \mathcal{L}_{\mathrm{basic}}$ labeled action transition, is in twofold related to a Chi 2.0 action transition:

- no counter reset of the hybrid time, i.e., $c > 0$

$$\begin{array}{l}
(p,\sigma) \xrightarrow{\left(\begin{array}{c} com_{\mathrm{time}}(t,c-1)|\big|_{v\in\mathcal{V}_{\mathrm{mCRL2}}} com_{\mathrm{mem}}^{[\![v]\!]}(w)|l| \\ diff(W)|\big|_{v\in\mathcal{V}_{\mathrm{mCRL2}}} com_{\mathrm{mem}'}^{[\![v]\!]}(w')|com_{\mathrm{time}'}(t,c) \end{array}\right)} time_{\mathrm{H}}(t,c')\ (p',\sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi,p'_\chi:P_{\mathrm{proc}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \overset{\sigma_\chi,l,W,\sigma'_\chi}{\longrightarrow} \langle p'_\chi, \sigma'_\chi \rangle
\end{array}$$

$$\begin{array}{rl}
\text{where} & \forall_{v\in\mathcal{V}_{\mathrm{mCRL2}}} \sigma_\chi(v) = w \overset{\triangle}{=} com_{\mathrm{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v\in\mathcal{V}_{\mathrm{mCRL2}}} \sigma'_\chi(v) = w' \overset{\triangle}{=} com_{\mathrm{mem}'}^{[\![v]\!]}(w') \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \overset{\triangle}{=} t
\end{array}$$

- a counter reset of the hybrid time:

$$\begin{array}{l}
(p,\sigma) \xrightarrow{\left(\begin{array}{c} com_{\mathrm{time}}(t,c)|\big|_{v\in dom(\sigma)} com_{\mathrm{mem}}^{[\![v]\!]}(w)|l| \\ diff(W)|\big|_{v\in dom(\sigma')} com_{\mathrm{mem}'}^{[\![v]\!]}(w')|com_{\mathrm{time}'}(t',0) \end{array}\right)} time_{\mathrm{H}}(t',0)\ (p',\sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi,p'_\chi:P_{\mathrm{proc}}} \exists_{d:\mathbb{R}^{>0}} \exists_{\theta_n,\theta_s,\theta_r:\mathcal{T}\mapsto val_{\mathrm{ah}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \overset{d,\sigma_\chi,(\theta_n,\theta_s,\theta_r)}{\longmapsto} \langle p_\chi, \sigma'_\chi \rangle \overset{\sigma'_\chi,l,W,\sigma'_\chi}{\longrightarrow} \langle p'_\chi, \sigma''_\chi \rangle
\end{array}$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = \sigma'_\chi(v) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \triangleq com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma''_\chi(v) = w' \triangleq com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & d = t' - t \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \triangleq t \\
\text{and} & \sigma''_\chi(\mathsf{time}) \triangleq t'
\end{array}
$$

Communication: An mCRL2 action transition that models the successful communication of a value e over channel $h \in \mathcal{H}$ is in twofold related to a Chi 2.0 action transition:

- no counter reset of the hybrid time, i.e., $c > 0$

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{\left(\begin{array}{c} com_{\text{time}}(t,c-1)|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}}^{[\![v]\!]}(w)|comm_{[\![h]\!]}(e)| \\ diff(W')|diff(W'')|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}'}^{[\![v]\!]}(w')|com_{\text{time}'}(t,c) \end{array}\right)}_{time_{\text{H}}(t,c')} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \xrightarrow{\sigma_\chi, h!?e, W, \sigma'_\chi} \langle p'_\chi, \sigma'_\chi \rangle,
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \triangleq com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma'_\chi(v) = w' \triangleq com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \triangleq t \\
\text{and} & \sigma_\chi, h!?e \triangleq comm_{[\![h]\!]}(e) \\
\text{and} & W \triangleq W' \cup W''
\end{array}
$$

- a counter reset of the hybrid time:

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{\left(\begin{array}{c} com_{\text{time}}(t,c)|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}}^{[\![v]\!]}(w)|comm_{[\![h]\!]}(e)| \\ diff(W')|diff(W'')|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}'}^{[\![v]\!]}(w')|com_{\text{time}'}(t',0) \end{array}\right)}_{time_{\text{H}}(t',0)} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \exists_{d:\mathbb{R}^{>0}} \exists_{\theta_n, \theta_s, \theta_r : \mathcal{T} \mapsto val_{\text{ah}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \xrightarrow{d, \sigma_\chi, (\theta_n, \theta_s, \theta_r)} \langle p_\chi, \sigma'_\chi \rangle \xrightarrow{\sigma'_\chi, h!?e, W, \sigma'_\chi} \langle p'_\chi, \sigma''_\chi \rangle
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = \sigma'_\chi(v) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \triangleq com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma''_\chi(v) = w' \triangleq com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & d = t' - t \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \triangleq t \\
\text{and} & \sigma''_\chi(\mathsf{time}) \triangleq t' \\
\text{and} & \sigma_\chi, h!?e \triangleq comm_{[\![h]\!]}(e) \\
\text{and} & W \triangleq W' \cup W''
\end{array}
$$

Internal actions: Relating mCRL2 action transitions to Chi 2.0 internal transitions. An mCRL2 action transition that models an internal action is in fourfold related to a Chi 2.0 action transition:

- no counter reset of the hybrid time, i.e., $c > 0$, and the internal action does not originate from a successful communication.

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{\left(\begin{array}{c} com_{\text{time}}(t, c-1) \big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}}^{[\![v]\!]}(w) | diff(W) | \\ \big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}'}^{[\![v]\!]}(w') | com_{\text{time}'}(t, c) \end{array}\right)} {}_{time_{\text{H}}(t, c')} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \xrightarrow{\sigma_\chi, \tau, W, \sigma'_\chi} \langle p'_\chi, \sigma'_\chi \rangle
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \triangleq com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma'_\chi(v) = w' \triangleq com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \triangleq t
\end{array}
$$

- no counter reset of the hybrid time, i.e., $c > 0$, and the internal action originates from a successful communication.

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{\left(\begin{array}{c} com_{\text{time}}(t, c-1) \big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}}^{[\![v]\!]}(w) | \\ diff(W') | diff(W'') \big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}'}^{[\![v]\!]}(w') | com_{\text{time}'}(t, c) \end{array}\right)} {}_{time_{\text{H}}(t, c')} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \xrightarrow{\sigma_\chi, \tau, W, \sigma'_\chi} \langle p'_\chi, \sigma'_\chi \rangle,
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \stackrel{\triangle}{=} com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma'_\chi(v) = w' \stackrel{\triangle}{=} com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \stackrel{\triangle}{=} t \\
\text{and} & W \stackrel{\triangle}{=} W' \cup W''
\end{array}
$$

- a counter reset of the hybrid time and the internal action does not originate from a successful communication.

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{com_{\text{time}}(t,c)|\big|_{v \in dom(\sigma)} com_{\text{mem}}^{[\![v]\!]}(w)|diff(W)|\big|_{v \in dom(\sigma')} com_{\text{mem}'}^{[\![v]\!]}(w')|com_{\text{time}'}(t',0)} {}_{time_{\text{H}}(t',0)} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \exists_{d:\mathbb{R}^{>0}} \exists_{\theta_n, \theta_s, \theta_r : \mathcal{T} \mapsto val_{\text{ah}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \overset{d, \sigma_\chi, (\theta_n, \theta_s, \theta_r)}{\longmapsto} \langle p_\chi, \sigma'_\chi \rangle \overset{\sigma'_\chi, \tau, W, \sigma'_\chi}{\longrightarrow} \langle p'_\chi, \sigma''_\chi \rangle
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = \sigma'_\chi(v) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \stackrel{\triangle}{=} com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma''_\chi(v) = w' \stackrel{\triangle}{=} com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & d = t' - t \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \stackrel{\triangle}{=} t \\
\text{and} & \sigma''_\chi(\mathsf{time}) \stackrel{\triangle}{=} t'
\end{array}
$$

- a counter reset of the hybrid time and the internal action originates from a successful communication.

$$
\begin{array}{l}
(p, \sigma) \xrightarrow{\left(\begin{array}{c} com_{\text{time}}(t,c)|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}}^{[\![v]\!]}(w)|diff(W')| \\ diff(W'')|\big|_{v \in \mathcal{V}_{\text{mCRL2}}} com_{\text{mem}'}^{[\![v]\!]}(w')|com_{\text{time}'}(t',0) \end{array} \right)} {}_{time_{\text{H}}(t',0)} (p', \sigma') \\
\Rightarrow \\
\quad \exists_{E:\mathcal{E}} \exists_{p_\chi, p'_\chi : P_{\text{proc}}} \exists_{d:\mathbb{R}^{>0}} \exists_{\theta_n, \theta_s, \theta_r : \mathcal{T} \mapsto val_{\text{ah}}} \\
\qquad E \Vdash \langle p_\chi, \sigma_\chi \rangle \overset{d, \sigma_\chi, (\theta_n, \theta_s, \theta_r)}{\longmapsto} \langle p_\chi, \sigma'_\chi \rangle \overset{\sigma'_\chi, \tau, W, \sigma'_\chi}{\longrightarrow} \langle p'_\chi, \sigma''_\chi \rangle
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = \sigma'_\chi(v) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = w \stackrel{\triangle}{=} com_{\text{mem}}^{[\![v]\!]}(w) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma''_\chi(v) = w' \stackrel{\triangle}{=} com_{\text{mem}'}^{[\![v]\!]}(w') \\
\text{and} & d = t' - t \\
\text{and} & \sigma_\chi(\mathsf{time}) = \sigma'_\chi(\mathsf{time}) \\
\text{and} & \sigma_\chi(\mathsf{time}) \stackrel{\triangle}{=} t \\
\text{and} & \sigma''_\chi(\mathsf{time}) \stackrel{\triangle}{=} t' \\
\text{and} & W \stackrel{\triangle}{=} W' \cup W''
\end{array}
$$

Idle transitions

mCRL2 idle transitions, i.e., the progression of time, are related to the continuous behavior for a Chi 2.0 specification in the following way:

$$
\begin{array}{l}
(p, \sigma) \rightsquigarrow_t \\
\Rightarrow \\
\quad \exists_{\theta_n, \theta_s, \theta_r : \mathcal{T} \mapsto val_{\text{ah}}} \exists_{p_\chi : P_{\text{proc}}} \exists_{E : \mathcal{E}} \; E \Vdash \langle p_\chi, \sigma_\chi \rangle \overset{d, \sigma_\chi, (\theta_n, \theta_s, \theta_r)}{\longmapsto} \langle p_\chi, \sigma'_\chi \rangle
\end{array}
$$

$$
\begin{array}{rl}
\text{where} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) = \sigma'_\chi(v) \\
\text{and} & \forall_{v \in \mathcal{V}_{\text{mCRL2}}} \sigma_\chi(v) \stackrel{\triangle}{=} \sigma(v) \\
\text{and} & \sigma_\chi(\mathsf{time}) \stackrel{\triangle}{=} \sigma(\mathsf{time}) \\
\text{and} & \sigma'_\chi(\mathsf{time}) \stackrel{\triangle}{=} t \\
\text{and} & d \stackrel{\triangle}{=} t - \sigma(\mathsf{time})
\end{array}
$$

A.2 Correspondence Relation between TSS and LPS

A.2.1 Labeled Transition System Associated with a Linear Process Specification

This subsection describes how a Labeled Transition System (LTS) can be associated with a Linear Process Specification (LPS) [Mon05].

Definition A.2.1 (Labeled Transition System)**.** A Labeled Transition System (LTS) is a triple $(S, L, \rightarrow, s_0)$ where:

- S is a set of states,
- L is a set of labels,
- $\longrightarrow \subseteq S \times L \times S$ is a transition relation,

- $s_0 \in S$ is the initial state.

An LPS describes a transition relation through the following Transition System Specification (TSS). The signature of the transition system specification is left implicit. The only deduction rule is the following

$$\frac{\models c_i(d, e_i) = true}{X(d) \xrightarrow{a(d,e_i)} X(g_i(d, e_i))}$$

where $\models c_i(d, e_i)$ indicates that the Boolean expression $c_i(d, e_i)$ must be derivable equal to *true*.

The LTS associated with closed term $X(p)$ with p a closed term of sort $\mathcal{P}$ is that part of the transition relation described by the LPS for X that is reachable from $X(p)$.

A.2.2 Labeled Transition System Associated with a Transition System Specification

[GV92] clearly defines how a transition relation is defined through a Transition System Specification (TSS). The Labeled Transition System (LTS) associated with a closed term $p \in \mathcal{C}(\Sigma)$ is obtained by considering that part of the transition relation described by the TSS that is reachable from p.

A.2.3 Lemmas

Lemma A.2.2. *For all $x \in \mathcal{V}$, $t \in \mathcal{T}(\Sigma)$, and substitutions $\mathbb{S}\colon \mathcal{V} \to \mathcal{T}(\Sigma)$*

$$x \in vars(t) \qquad \Rightarrow \qquad \mu_x^t(\mathbb{S}(t)) = \mathbb{S}(x)$$

Proof. By induction on the structure of term t.

- t is a variable. In case $t = x$ we have $\mu_x^t(\mathbb{S}(t)) = \mu_x^x(\mathbb{S}(t)) = \mathbb{S}(x)$. The case where t is a variable different from x cannot occur as $x \notin vars(t)$.

- t is of the form $f(t_1, \ldots, t_{ar(f)})$ for some $f \in \Sigma$ and $t_1, \ldots, t_{ar(f)} \in \mathcal{T}(\Sigma)$. Since $x \in vars(t)$, we have $x \in vars(t_i)$ for some i such that $1 \leq i \leq ar(f)$. By induction hypothesis we obtain $\mu_x^{t_i}(\mathbb{S}(t_i)) = \mathbb{S}(x)$. Note that since $x \in vars(t_i)$ we have the equation $\mu_x^t(\mathbb{S}(t)) = \mu_x^{t_i}(\pi_i(\mathbb{S}(t)))$. Since $\pi_i(\mathbb{S}(t)) = \pi_i(\mathbb{S}(f(t_1, \ldots, t_{ar(f)}))) = \pi_i(f(\mathbb{S}(t_1), \ldots, \mathbb{S}(t_{ar(f)}))) = \mathbb{S}(t_i)$, we then also have $\mu_x^t(\mathbb{S}(t)) = \mu_x^{t_i}(\pi_i(\mathbb{S}(t))) = \mu_x^{t_i}(\mathbb{S}(t_i)) = \mathbb{S}(t_i)$, which was to be shown.

□

Lemma A.2.3. *For all $t \in \mathcal{T}(\Sigma)$, and substitutions $\mathbb{S}\colon \mathcal{V} \to \mathcal{T}(\Sigma)$*

$$\sigma^t(\mathbb{S}(t)) = true$$

Proof. By induction on the structure of term t.

1. t is a variable, say x. Then $\sigma^t(\mathbb{S}(t)) = \sigma^x(\mathbb{S}(x)) = true$.
2. t is of the form $f(t_1, \ldots, t_{ar(f)})$ for some $f \in \Sigma$ and $t_1, \ldots, t_{ar(f)} \in \mathcal{T}(\Sigma)$. By induction hypothesis we have $\sigma^{t_i}(\mathbb{S}(t_i)) = true$ for all i such that $1 \leq i \leq ar(f)$.

   Then $\sigma^t(\mathbb{S}(t)) = is_f(f(t_1, \ldots, t_{ar(f)})) \wedge \bigwedge_{i=1}^{ar(f)} \sigma^{t_i}(\pi_i(\mathbb{S}(f(t_1, \ldots, t_{ar(f)})))) = true \wedge \bigwedge_{i=1}^{ar(f)} \sigma^{t_i}(\mathbb{S}(t_i)) = true$.

□

Lemma A.2.4. *For all $t \in \mathcal{T}(\Sigma)$ and $p \in \mathcal{C}(\Sigma)$ such that $\sigma^t(p)$. If $\mu_x^t(p) = \mathbb{S}(x)$ for all $x \in vars(t)$, then $\mathbb{S}(t) = p$.*

Proof. By induction on the structure of term t. Assume that $\mu_x^t(p) = \mathbb{S}(x)$ for all $x \in vars(t)$.

1. t is a variable, say x. Since $\mu_x^t(p) = \mathbb{S}(x)$ by assumption, and $\mu_x^t(p) = \mu_x^x(p) = p$ we have $\mathbb{S}(t) = \mathbb{S}(x) = \mu_x^t(p) = p$.
2. t is of the form $f(t_1, \ldots, t_{ar(f)})$ for some $f \in \Sigma$ and $t_1, \ldots, t_{ar(f)} \in \mathcal{T}(\Sigma)$.

   From $\sigma^t(p)$ it follows that $\sigma^{t_i}(\pi_i(p))$ for all $1 \leq i \leq ar(f)$.

   From the assumption that $\mu_x^t(p) = \mathbb{S}(x)$ and the fact that $\mu_x^t(p) = \mu_x^{f(t_1,\ldots,t_{ar(f)})}(p) = \mu_x^{t_i}(\pi_i(p))$ for those t_i in which x occurs it follows that $\mu_x^{t_i}(\pi_i(p)) = \mathbb{S}(x)$ for all variables $x \in vars(t_i)$. Hence, by induction hypothesis we have $\mathbb{S}(t_i) = \pi_i(p)$ for all $1 \leq i \leq ar(f)$. Then we have, $\mathbb{S}(t) = \mathbb{S}(f(t_1, \ldots, t_{ar(f)})) = f(\mathbb{S}(t_1), \ldots, \mathbb{S}(t_{ar(f)})) = f(\pi_1(p), \ldots, \pi_{ar(f)}(p)) = p$. Note that we have used that $is_f(p)$ implies $f(\pi_1(p), \ldots, \pi_{ar(f)}(p)) = p$ for all $p \in \mathcal{C}(\Sigma)$. This is easily proven by induction on the structure of p.

□

A.2.4 Proof of the Correspondence Theorem

Theorem A.2.5. *Let $(\Sigma, \mathcal{D})$ be an mCRL2-restrictive* TSS *in the* De Simone *format. Then for every $p \in \mathcal{C}(\Sigma)$, the* LTS *associated with p and the* LTS *associated with $X(p)$ are isomorphic.*

Proof. Obviously, it suffices to show that for all $p, p' \in \mathcal{C}(\Sigma)$ and $l \in \mathcal{A}$

$$p \xrightarrow{l} p' \quad \Rightarrow \quad relation(l, p') \in R(p) \tag{1}$$

and

$$relation(l, p') \in R(p) \quad \Rightarrow \quad p \xrightarrow{l} p' \tag{2}$$

since $relation(l, p') \in R(p)$ iff $X(p) \xrightarrow{l} X(p')$ follows directly from the semantics of an LPS.

We give a proof for equation 1. We prove this part by induction on the depth of the proof tree of $p \xrightarrow{l} p'$. Now assume that the last step in this proof tree is the application of deduction rule $d \in \mathcal{D}$ of the form

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)})}[Cond_d]$$

and let $\mathbb{S}$ be a substitution such that $\mathbb{S}(f(x_1, \ldots, x_{ar(f)})) = p$, $\mathbb{S}(t) = p'$, $\mathbb{S}(x_i) \xrightarrow{l_i} \mathbb{S}(y_i)$ for all $i \in I$, and $Cond_d$.

1. $is_f(p) = is_f(\mathbb{S}(f(x_1, \ldots, x_{ar(f)}))) = is_f(f(\mathbb{S}(x_1), \ldots, \mathbb{S}(x_{ar(f)}))) = true$
2. $\sigma^t(p') = \sigma^t(\mathbb{S}(t)) = true$. The last step is due to Lemma A.2.3.
3.
   - $Cond_d$ holds for the labels of the premises and the label for the conclusion.
   - For every $i \in I$ with $y_i \in vars(t)$: $\mu^t_{y_i}(p') = \mu^t_{y_i}(\mathbb{S}(t)) = \mathbb{S}(y_i)$ according to Lemma A.2.2.

     Note that $\pi_i(p) = \pi_i(\mathbb{S}(f(x_1, \ldots, x_{ar(f)}))) = \pi_i(f(\mathbb{S}(x_1), \ldots, \mathbb{S}(x_{ar(f)}))) = \mathbb{S}(x_i)$. By induction hypothesis, for each $i \in I$, $relation(l_i, \mathbb{S}(y_i)) \in R(\mathbb{S}(x_i))$. Therefore,

     $$\bigwedge_{i \in I} y_i \in vars(t) \qquad \Rightarrow \qquad relation(l_i, \mu^t_{y_i}(p')) \in R(\pi_i(p))$$

   - For every $i \in I$ with $y_i \notin vars(t)$, by induction hypothesis, $relation(l_i, \mathbb{S}(y_i)) \in R(\mathbb{S}(x_i))$. As before $\mathbb{S}(x_i) = \pi_i(p)$. Therefore,

     $$\bigwedge_{i \in I} y_i \notin vars(t) \qquad \Rightarrow \qquad \exists_{z_i} relation(l_i, z_i) \in R(\pi_i(p))$$

4. For every j such that $1 \leq j \leq ar(f)$ and $j \notin I$ and $x_j \in vars(t)$ we have

   $$\mu^t_{x_j}(p') = \mu^t_{x_j}(\mathbb{S}(t)) = \mathbb{S}(x_j) = \pi_j(p)$$

   using Lemma A.2.2. Therefore,

   $$\bigwedge_{j \notin I} x_j \in vars(t) \qquad \Rightarrow \qquad \mu^t_{x_j}(p') = \pi_j(p)$$

From this we can conclude that $relation(l, p') \in R_d(p)$ and therefore also $relation(l, p') \in R(p)$.

Next we prove equation 2 by induction on closed term p. Assume that $X(p) \xrightarrow{l} X(p')$. Then this must be due to the fact that $relation(l, p') \in R(p)$. By definition this means that there exists a deduction rule $d \in \mathcal{D}$ such that $relation(l, p') \in R_d(p)$.

As d is a deduction rule in De Simone format, iff it is of the form

$$\frac{\{x_i \xrightarrow{l_i} y_i \mid i \in I\}}{f(x_1, \ldots, x_{ar(f)})}[Cond_d]$$

From the definition of R_d it follows that there exist q_i, for $i \in I$ with $y_i \notin vars(t)$ such that $relation(l_i, q_i) \in R(\pi_i(p))$. Now, define a substitution $\mathbb{S}$ such that

- $\mathbb{S}(x_i) = \pi_i(p)$ for $1 \leq i \leq ar(f)$,
- $\mathbb{S}(y_i) = \mu^t_{y_i}(p')$ for $i \in I$ such that $y_i \in vars(t)$,
- $\mathbb{S}(y_i) = q_i$ for $i \in I$ such that $y_i \notin vars(t)$.

Now we can establish the following facts:

1. $Cond_d$ holds for the labels of the premises and the label for the conclusion.
2. $\mathbb{S}(f(x_1, \ldots, x_{ar(f)})) = f(\mathbb{S}(x_1), \ldots, \mathbb{S}(x_{ar(f)})) = f(\pi_1(p), \ldots, \pi_{ar(f)}(p)) = p$. The last step follows from $is_f(p)$ (since $relation(l, p') \in R_d(p)$).
3. From $relation(l, p') \in R_d(p)$ it follows that $\mu^t_{x_j}(p') = \pi_j(p) = \mathbb{S}(x_j)$ for all $1 \leq j \leq ar(f)$ such that $j \notin I$. By definition of $\mathbb{S}$, for $i \in I$ and $y_i \in vars(t)$, also $\mu^t_{y_i}(p') = \mathbb{S}(y_i)$. Therefore, as $\sigma^t(p)$ also follows from $relation(l, p') \in R_d(p)$, by Lemma A.2.4 we have $\mathbb{S}(t) = p'$.
4. From $relation(l, p') \in R_d(p)$ it follows that $relation(l_i, \mu^t_{y_i}(p')) \in R(\pi_i(p))$, for $i \in I$ and $y_i \in vars(t)$ and $relation(l_i, q_i) \in R(\pi_i(p))$, for $i \in I$ and $y_i \notin vars(t)$. By induction we then have $\pi_i(p) \xrightarrow{l_i} \mu^t_{y_i}(p')$, for $i \in I$ and $y_i \in vars(t)$ and $\pi_i \xrightarrow{l_i} q_i$, for $i \in I$ and $y_i \notin vars(t)$. Since $\pi_i(p) = \mathbb{S}(x_i)$ and $\mu^t_{y_i}(p') = \mathbb{S}(y_i)$, the premises of the deduction rule d are all derivable.

We can conclude that $p \xrightarrow{l} p'$. □

Appendix B

Models

This chapter presents the (disclosed) mCRL2 source code models that have been used in this thesis.

B.1 2×2 Switch Models

This section presents the mCRL2 source code models for the 2×2 switch from Chapter 3. We present the models for the simplified switch (Chapter B.1.1), the original switch (Chapter B.1.2) and the modified switch (Chapter B.1.3).

B.1.1 The Simplified Switch

```
%
% Sorts
%
sort Bit    = struct b0 | b1;
     Packet = struct Pkg(data: Data);
     Data   = struct d0 | d1;

map dest : Packet -> Nat;
eqn dest(Pkg(d0)) = 0;
    dest(Pkg(d1)) = 1;

%
% Constant for size of FIFO queues
%
map cap: Pos;
eqn cap = 3;

%
% Action declarations
%
act send: Nat#Nat#Packet;
    recv: Nat#Nat#Packet;
    com:  Nat#Nat#Packet;
```

```
24      grant: Nat#Nat#Packet;
25      free: Nat#Nat#Packet;
26
27     % Remove element from OutputFIFO
28      send_ext: Nat#Packet;
29     % Add element to InputFIFO
30      recv_ext: Nat#Packet;
31
32  proc InputFIFO(ID: Nat, c: List(Packet)) =
33         % Packet with different destinations must be send simultaneous from
34         % different InputFIFOs
35           sum n: Nat. sum p: Packet.
36           ((n != ID) && (p != rhead(c)) && (c != []))
37           -> send(ID, dest(rhead(c)), rhead(c)) | free(n, dest(p), p)
38            . InputFIFO(ID, rtail(c))
39         % Grant prioritization to InputFIFO processes that have a lower rank ID,
40         % but have the same destination
41         + sum n: Nat.
42           ((n < ID) && (c != []))
43           -> grant(n, dest(rhead(c)), rhead(c)) . InputFIFO(ID, c)
44         % Take prioritization over InputFIFO processes that grant communication,
45         % which have a higher rank ID and have the same destination
46         + (c != []) ->
47                send(ID, dest(rhead(c)), rhead(c))
48              . InputFIFO(ID, rtail(c))
49         % Grant communication to all other InputFIFO processes
50         + sum n,m: Nat. sum p: Packet. (c == [])
51           -> grant(n, m, p) . InputFIFO(ID, c)
52         % Fill InputFIFO with packets
53         + (#c < cap) -> sum p: Packet. recv_ext(ID, p) . InputFIFO(ID, p |> c);
54
55  proc OutputFIFO(id: Nat, c: List(Packet)) =
56         % Fill OutputFIFO with packets if not over capacitated
57           (#c < cap) -> sum org: Nat. sum p:Packet. recv(org, id, p)
58                        . OutputFIFO(id, p |> c)
59         % Remove packets from OutputFIFO
60         + (c != [])  -> send_ext(id, rhead(c)). OutputFIFO(id, rtail(c));
61
62  init
63   block({send, recv, grant, free},
64   comm({send | recv | free -> com},
65   comm({send | recv | grant -> com},
66     InputFIFO(0,[]) ||
67     InputFIFO(1,[]) ||
68     OutputFIFO(0,[]) ||
69     OutputFIFO(1,[])
70   ))
71  );
```

B.1.2 The Original Switch

```
1   % ----------------
2   % Sorts
3   % ----------------
4   sort Bit    = struct zero | one;
5        Packet = struct Pkg(b1: Bit, i: Bool);
6   map  dest : Packet -> Nat;
7   var  i: Bool;
8   eqn  dest(Pkg(zero, i)) = 0;
9        dest(Pkg(one,  i)) = 1;
10
11  % ----------------------------------
12  % Constant for size of FIFO queues
13  % ----------------------------------
```

```
14  map cap: Pos;
15  eqn cap = 3;
16
17  % ------------------
18  % Action declarations
19  % ------------------
20  act inc;
21  act send: Nat#Nat#Packet;
22      recv: Nat#Nat#Packet;
23      com:  Nat#Nat#Packet;
24      grant:Nat#Nat#Packet;
25      free: Nat#Nat#Packet;
26
27      % Remove element from OutputFIFO
28      send_ext: Nat#Packet;
29      % Add element to InputFIFO
30      recv_ext: Nat#Packet;
31
32  proc InputFIFO(ID: Nat, c: List(Packet)) =
33          % Packet with different destinations must be send simultaneous from
34          % different InputFIFOs
35            sum p: Packet.
36            (c != [])
37            -> (((i(p) != i(rhead(c)) ||  (i(p) == i(rhead(c))) && !i(p))
38               && dest(p) != dest(rhead(c))) ->
39               (sum n:Nat. (n != ID)
40                 -> send(ID, dest(rhead(c)), rhead(c)) | free(n, dest(p), p)
41                  . InputFIFO(ID, rtail(c)))
42               +  !((i(p) != i(rhead(c)) ||  (i(p) == i(rhead(c)))
43                  && !i(p)) && dest(p) != dest(rhead(c))) -> (
44                        % Grant prioritization to InputFIFO processes that have
45                        % a lower rank ID and are both interesting
46                           sum l: Nat.(l < ID)
47                           -> grant(l, dest(p), p) . InputFIFO(ID, c)
48                        % Take prioritization over InputFIFO processes that grant
49                        % communication which have a higher rank ID and are both
50                        % interesting
51                        + send(ID, dest(rhead(c)), rhead(c))
52                           . InputFIFO(ID, rtail(c))
53                   )
54               )
55          % Grant prioritization to InputFIFO processes that have a lower rank ID,
56          % but have the same destination
57          + sum n: Nat.
58            sum j: Bool. % The value "interesting" is not important
59            ((n < ID) && (c != []) && !i(rhead(c)))
60            -> (grant(n, dest(rhead(c)), Pkg(b1(rhead(c)), j)) . InputFIFO(ID, c))
61          % Take prioritization over InputFIFO processes that grant communication,
62          % which have a higher rank ID and have the same destination
63          + (c != []) ->
64               (send(ID, dest(rhead(c)), rhead(c))
65               . InputFIFO(ID, rtail(c))
66               )
67          % Grant communication to all other InputFIFO processes
68          + sum n: Nat. sum p: Packet. (c == [])
69            -> (grant(n, dest(p), p ) . InputFIFO(ID, c))
70          % Fill InputFIFO with packets
71          + (#c < cap) -> (sum p: Packet. recv_ext(ID, p) . InputFIFO(ID, p |> c));
72
73  proc OutputFIFO(id: Nat, c: List(Packet)) =
74          % Fill OutputFIFO with packets if not over capacitated
75            (#c < cap) -> sum org: Nat. sum p: Packet.
76               ( !i(p) ->       recv(org, id, p) . OutputFIFO(id, p |> c)
77               +  i(p) -> inc | recv(org, id, p) . OutputFIFO(id, p |> c)
78               )
```

```
79          % Remove packets from OutputFIFO
80          + (c != [])  -> send_ext(id, rhead(c)) . OutputFIFO(id, rtail(c));
81
82  proc Input(i: Nat, c: List(Packet)) =
83            (#c < cap) -> sum p:Packet. recv_ext(i,p) . Input(i, p |> c)
84          + (c != [])  -> sum p:Packet.
85              ((dest(p) != dest(rhead(c))) && (!i(p) || !i(rhead(c))))
86             -> sum n:Nat. (n != i)
87               -> send(i, dest(rhead(c)), rhead(c)) | free(n, dest(p),p)
88                . Input(i, rtail(c))
89          + (c != [])  -> send(i, dest(rhead(c)), rhead(c)) . Input(i, rtail(c))
90          + (c == [])  -> sum n,m: Nat. sum p: Packet. grant(n,m,p) . Input(i,c)
91          + ((c != []) && i(rhead(c))) -> sum p:Packet.
92              ((dest(p) == dest(rhead(c))) || i(p))
93             -> sum n:Nat. (n < i) -> grant(n, dest(p), p) .Input(i,c)
94          + ((c != []) && !i(rhead(c))) -> sum p:Packet. (b1(p) == b1(rhead(c)))
95            -> sum n:Nat. (n < i) -> grant(n, dest(rhead(c)),p) . Input(i, c);
96
97  init
98   block({send, recv, grant, free},
99   comm({send | grant |recv -> com},
100  comm({send | free | recv -> com},
101     Input(0,[]) ||
102     Input(1,[]) ||
103     OutputFIFO(0,[]) ||
104     OutputFIFO(1,[])
105   ))
106 );
```

B.1.3 The Modified Switch

```
1   % ----------------
2   % Sorts
3   % ----------------
4   sort Bit    = struct zero | one;
5        Packet = struct Packet(b1: Bit, i: Bool);
6
7   map dest : Packet -> Nat;
8   var i: Bool;
9   eqn dest(Packet(zero, i)) = 0;
10      dest(Packet(one, i)) = 1;
11
12  % ------------------------------
13  % Constant for size of FIFO queues
14  % ------------------------------
15  map cap: Pos;
16  eqn cap = 3;
17
18  % --------------------
19  % Action declarations
20  % --------------------
21  act inc;
22  act send: Nat#Nat#Packet;
23      recv: Nat#Nat#Packet;
24      com:  Nat#Nat#Packet;
25      grant: Nat#Nat#Packet;
26      free: Nat#Nat#Packet;
27      % Remove element from OutputFIFO
28      leave: Nat#Packet;
29      % Add element to InputFIFO
30      enter: Nat#Packet;
31
32  proc InputFIFO(ID: Nat, c: List(Packet)) =
33          % Packet with different destinations must be send simultaneous from
```

```
34          % different InputFIFOs
35            sum p: Packet.
36            (c != [])
37            -> ( ((i(p) != i(rhead(c)) || (i(p) == i(rhead(c))) && !i(p))
38                    && dest(p) != dest(rhead(c))) -> (sum n:Nat. (n != ID)
39                    -> send(ID, dest(rhead(c)), rhead(c)) | free(n, dest(p), p)
40                     . InputFIFO(ID, rtail(c)))
41               + !((i(p) != i(rhead(c)) || (i(p) == i(rhead(c))) && !i(p))
42                  && dest(p) != dest(rhead(c))) -> (
43                       % Grant prioritization to InputFIFO processes that have
44                       % a lower rank ID and are both interesting
45                         sum l: Nat.
46                         (l > ID)
47                         -> grant(l, dest(p), p) . InputFIFO(ID, c)
48
49                       % Take prioritization over InputFIFO processes that
50                       % grant communication which have a higher rank ID and
51                       % are both interesting
52                       + send(ID, dest(rhead(c)), rhead(c)). InputFIFO(ID, rtail(c))
53                     )
54               )
55          % Grant prioritization to InputFIFO processes that have a lower rank
56          % ID, but have the same destination
57          + sum n: Nat.
58            sum i: Bool. % The value "interesting" is not important
59            ((n < ID) && (c != []) && !i(rhead(c)))
60            -> grant(n, dest(rhead(c)), Packet(b1(rhead(c)),i)) . InputFIFO(ID, c)
61          % Take prioritization over InputFIFO processes that grant communication,
62          % which have a higher rank ID and have the same destination
63          + (c != []) -> send(ID, dest(rhead(c)), rhead(c)) . InputFIFO(ID, rtail(c))
64          % Grant communication to all other InputFIFO processes
65          + sum n,m: Nat. sum p: Packet. (c == [])
66            -> grant(n, m, p) . InputFIFO(ID, c)
67          % Fill InputFIFO with packets
68          + (#c < cap) -> sum p: Packet. enter(ID, p) . InputFIFO(ID, p |> c);
69
70  proc Input(i: Nat, c: List(Packet)) =
71            (#c < cap) -> sum p:Packet. enter(i,p) . Input(i, p |> c)
72          + (c != []) -> sum p:Packet. ((dest(p) != dest(rhead(c))) &&
73              (!i(p) || !i(rhead(c)))) ->
74              sum n:Nat. (n != i)
75              -> send(i, dest(rhead(c)), rhead(c)) | free(n, dest(p),p)
76               . Input(i, rtail(c))
77          + (c != []) -> send(i, dest(rhead(c)), rhead(c)) . Input(i, rtail(c))
78          + (c == []) -> sum n,m: Nat. sum p: Packet. grant(n,m,p) . Input(i,c)
79          + ((c != []) && i(rhead(c))) -> sum p:Packet. (
80            (dest(p) == dest(rhead(c)))) ->
81            sum n:Nat. (n < i) -> grant(n, dest(p), p) . Input(i,c)
82          + ((c != []) && i(rhead(c))) -> sum p:Packet. (
83              (dest(p) != dest(rhead(c))) && i(p))
84              -> sum n:Nat. (n > i) -> grant(n, dest(p), p) . Input(i,c)
85          + ((c != []) && !i(rhead(c))) -> sum p:Packet. (b1(p) == b1(rhead(c)))
86            -> sum n:Nat. (n < i) -> grant(n, dest(rhead(c)),p) . Input(i, c);
87
88  proc OutputFIFO(id: Nat, c: List(Packet)) =
89         % Fill OutputFIFO with packets if not over capacitated
90           (#c < cap) -> sum org: Nat. sum p: Packet.
91              ( !i(p) ->       recv(org, id, p) . OutputFIFO(id, p |> c)
92              +  i(p) -> inc | recv(org, id, p) . OutputFIFO(id, p |> c)
93              )
94         % Remove packets from OutputFIFO
95         + (c != []) -> leave(id, rhead(c)) . OutputFIFO(id, rtail(c));
96
97  init
98   block({send, recv, grant, free},
```

```
 comm({send | recv | grant -> com},
 comm({send | recv | free -> com},
   Input(0,[]) ||
   Input(1,[]) ||
   OutputFIFO(0,[]) ||
   OutputFIFO(1,[])
 ))
);
```

B.2 Translated Chi 2.0 Models

This section presents the four mCRL2 source code models for the Chi 2.0 examples that have been constructed using the translation from Chapter 5.

B.2.1 Guarded Action Update Example

```
sort ChiLabelsBasic = struct a ;

sort Variables = struct time;

sort TimeSort = Nat;

sort Htime = struct htime(pi_time: TimeSort, pi_counter: TimeSort);

act a;
act ctau;
act SendAbsTime, RecvAbsTime, CommAbsTime,
    SendAbsUpdTime, RecvAbsUpdTime, CommAbsUpdTime: Htime;
    SendMem_s, RecvMem_s, CommMem_s : Bool;
    SendUpdMem_s, RecvUpdMem_s, CommUpdMem_s : Bool;
    chng: Set(Variables);

map Urgent_G: ChiLabelsBasic -> Bool;
eqn Urgent_G(a) = true;

proc ProcTime(t: Htime) = sum t': Htime. pred_htime(t, t')
                          -> SendAbsTime(t) | RecvAbsUpdTime(t'). ProcTime(t');

map pred_htime: Htime#Htime -> Bool;
var t, t': Htime;
eqn pred_htime(t, t') = (t' == htime(pi_time(t), pi_counter(t)+1)) ||
                        (pi_time(t') > pi_time(t) && pi_counter(t') == 0);

proc ProcChi =
  sum t: Htime. sum t': Htime.
  ((lambda time': TimeSort, time: TimeSort, predicate: TimeSort -> Bool.
     (predicate(time') &&
      if(Urgent_G(a),
        forall x: TimeSort. time <= x && x < time' => !(predicate(x)), true)))
    (pi_time(t'), pi_time(t), (lambda time: TimeSort. time >= 1))) ->
       (RecvAbsTime(t) | a | chng({time}) | SendAbsUpdTime(t'));

init hide({ctau},
      allow({CommAbsTime | a | chng | CommAbsUpdTime,
             CommAbsTime | ctau | chng | CommAbsUpdTime
            },
       comm({ SendAbsTime | RecvAbsTime -> CommAbsTime,
              SendAbsUpdTime | RecvAbsUpdTime -> CommAbsUpdTime,
              SendMem_s | RecvMem_s -> CommMem_s,
              SendUpdMem_s | RecvUpdMem_s -> CommUpdMem_s
            },
```

```
46          ProcTime(htime(0, 0)) || ProcChi)));
```

B.2.2 Alternative Composition Example

```
1   sort ChiLabelsBasic = struct a | b;
2
3   sort TimeSort = Nat;
4
5   sort Variables = struct s | time;
6
7   sort Htime = struct htime(pi_time: TimeSort, pi_counter: Nat);
8
9   act a, b;
10  act ctau;
11  act SendAbsTime, RecvAbsTime, CommAbsTime,
12      SendAbsUpdTime, RecvAbsUpdTime, CommAbsUpdTime: Htime;
13      SendMem_s, RecvMem_s, CommMem_s: Bool;
14      SendUpdMem_s, RecvUpdMem_s, CommUpdMem_s: Bool;
15      chng: Set(Variables);
16
17  map Urgent_G: ChiLabelsBasic -> Bool;
18  eqn Urgent_G(a) = true;
19      Urgent_G(b) = false;
20
21  sort ChiProcessTerm =
22     struct ChiAlt(p_1: ChiProcessTerm, p_2: ChiProcessTerm)
23          | ChiActionUpdate(pi_guard: Bool#Bool#TimeSort -> Bool,
24                            pi_a: ChiLabelsBasic,
25                            pi_chng: Set(Variables),
26                            pi_r: Bool#Bool#TimeSort -> Bool);
27
28  map CompMaxDelay: ChiProcessTerm#TimeSort#Bool -> TimeSort;
29  var p_1, p_2: ChiProcessTerm;
30      u: Bool#Bool#TimeSort-> Bool;
31      t: TimeSort;
32      s, s': Bool;
33      w: Set(Variables);
34      r: Bool#Bool#TimeSort -> Bool;
35  eqn CompMaxDelay(ChiAlt(p_1, p_2), t, s) =
36                            min(CompMaxDelay(p_1, t, s), CompMaxDelay(p_2, t, s));
37      % Shortcut (Max time in model is 1000)
38      CompMaxDelay(ChiActionUpdate(u, a, w, r), t, s) = if(Urgent_G(a), 2, 1000);
39      CompMaxDelay(ChiActionUpdate(u, b, w, r), t, s) = if(Urgent_G(b), 1, 10);
40
41  proc ProcTime(t: Htime) = sum t': Htime. pred_htime(t, t')
42                            -> SendAbsTime(t) | RecvAbsUpdTime(t'). ProcTime(t');
43
44  map pred_htime: Htime#Htime -> Bool;
45  var t, t': Htime;
46  eqn pred_htime(t, t') = (t' == htime(pi_time(t), pi_counter(t)+1)) ||
47                          (pi_time(t') > pi_time(t) && pi_counter(t') == 0);
48
49  proc ProcMem(s: Bool) =
50      sum s': Bool. SendMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
51    +               SendMem_s(s) | CommUpdMem_s(s ) . ProcMem(s )
52    + sum s': Bool. CommMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
53    +               CommMem_s(s) | CommUpdMem_s(s ) . ProcMem(s );
54
55  glob dc: Bool;
56  proc ProcChi =
57    sum t: Htime. sum t': Htime. sum v1: Bool. sum w1: Bool.
58      (pi_time(t') - pi_time(t) <=
59        CompMaxDelay(
60          ChiAlt(
```

```
61            ChiActionUpdate(
62              lambda v1: Bool, s2: Bool, time: TimeSort. time >= 2, a, {},
63              lambda v1: Bool, s2: Bool, time: TimeSort. true),
64          ChiActionUpdate(
65              lambda v1: Bool, s2: Bool, time: TimeSort. time <= 10 && time >= 1, b, {s},
66              lambda v1: Bool, s2: Bool, time: TimeSort. s2 == false)),
67       pi_time(t), v1)) ->
68            comm {RecvAbsTime | RecvAbsTime -> RecvAbsTime,
69                  SendAbsUpdTime | SendAbsUpdTime -> SendAbsUpdTime,
70                  RecvMem_s | RecvMem_s -> RecvMem_s,
71                  SendUpdMem_s | SendUpdMem_s -> SendUpdMem_s},
72   (sum t: Htime. sum t': Htime. sum w1: Bool.
73   ((lambda v1: Bool, time': TimeSort, time: TimeSort,
74       predicate: Bool#TimeSort -> Bool.
75        (predicate(v1, time') &&
76         if(Urgent_G(a),
77            forall x: TimeSort. time <= x && x < time' => !(predicate(v1, x)),
78            true)))
79      (dc, pi_time(t'), pi_time(t), (lambda v1: Bool, time: TimeSort. time >= 2))) ->
80             ( RecvAbsTime(t) | a | chng({time})
81             | SendAbsUpdTime(t')@pi_time(t')
82             ) | RecvAbsTime(t) | SendAbsUpdTime(t'))
83 + (sum t: Htime.sum t': Htime. sum w1: Bool.
84   ((lambda v1: Bool, time': TimeSort, time: TimeSort,
85       predicate: Bool#TimeSort -> Bool.
86         (predicate(v1, time') &&
87          if(Urgent_G(b),
88             forall x: TimeSort. time <= x && x < time' => !(predicate(v1, x)),
89             true)))
90      (dc, pi_time(t'), pi_time(t),
91        (lambda v1: Bool, time: TimeSort. time <= 10 && time >= 1)) &&
92      (lambda w1: Bool, time': TimeSort, time: TimeSort,
93       predicate: Bool#TimeSort -> Bool.
94         (predicate(w1, time')))
95      (w1, pi_time(t'), pi_time(t),
96         (lambda w1: Bool, time: TimeSort. w1 == false))) ->
97            ( SendUpdMem_s(w1) | RecvAbsTime(t) | b | chng({s, time})
98            | SendAbsUpdTime(t')@pi_time(t')
99            ) | RecvMem_s(v1) | RecvAbsTime(t) | SendAbsUpdTime(t')));
100
101 init hide({ctau},
102       allow({CommAbsTime | CommMem_s | a | chng | CommUpdMem_s | CommAbsUpdTime,
103              CommAbsTime | CommMem_s | b | chng | CommUpdMem_s | CommAbsUpdTime,
104              CommAbsTime | CommMem_s | ctau | chng |  CommUpdMem_s | CommAbsUpdTime,
105             },
106        comm({ SendAbsTime | RecvAbsTime -> CommAbsTime,
107               SendAbsUpdTime | RecvAbsUpdTime -> CommAbsUpdTime,
108               SendMem_s | RecvMem_s -> CommMem_s,
109               SendUpdMem_s | RecvUpdMem_s -> CommUpdMem_s
110             },
111        ProcMem(true) || ProcTime(htime(0, 0)) || ProcChi)));
```

B.2.3 Parallel Composition Example

```
1  sort ChiLabelsBasic = struct a | b;
2
3  sort TimeSort = Nat;
4
5  sort Variables = struct s | time;
6
7  sort Htime = struct htime(pi_time: TimeSort, pi_counter: Nat);
8
9  act a, b;
10 act ctau;
```

```
11  act SendAbsTime, RecvAbsTime, CommAbsTime,
12      SendAbsUpdTime, RecvAbsUpdTime, CommAbsUpdTime: Htime;
13      SendMem_s, RecvMem_s, CommMem_s: Bool;
14      SendUpdMem_s, RecvUpdMem_s, CommUpdMem_s: Bool;
15      chng: Set(Variables);
16
17  map Urgent_G: ChiLabelsBasic -> Bool;
18  eqn Urgent_G(a) = true;
19      Urgent_G(b) = false;
20
21  proc ProcTime(t: Htime) = sum t':Htime. pred_htime(t, t')
22                            -> SendAbsTime(t) | RecvAbsUpdTime(t'). ProcTime(t');
23
24  map pred_htime: Htime#Htime -> Bool;
25  var t, t': Htime;
26  eqn pred_htime(t, t') = (t' == htime(pi_time(t), pi_counter(t)+1)) ||
27                          (pi_time(t') > pi_time(t) && pi_counter(t') == 0);
28
29  proc ProcMem(s: Bool) =
30      sum s': Bool . SendMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
31    +                SendMem_s(s) | CommUpdMem_s(s ) . ProcMem(s )
32    + sum s': Bool . CommMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
33    +                CommMem_s(s) | CommUpdMem_s(s ) . ProcMem(s );
34
35  glob dc: Bool;
36  proc ProcChi =
37    allow(
38    {
39                                  RecvAbsTime | a | chng | SendAbsUpdTime,
40                                  RecvAbsTime | b | chng | SendAbsUpdTime,
41                                  RecvAbsTime | ctau | chng | SendAbsUpdTime,
42
43      RecvMem_s                 | RecvAbsTime | a | chng | SendAbsUpdTime,
44      RecvMem_s                 | RecvAbsTime | b | chng | SendAbsUpdTime,
45      RecvMem_s                 | RecvAbsTime | ctau | chng | SendAbsUpdTime,
46
47                  SendUpdMem_s | RecvAbsTime | a | chng | SendAbsUpdTime,
48                  SendUpdMem_s | RecvAbsTime | b | chng | SendAbsUpdTime,
49                  SendUpdMem_s | RecvAbsTime | ctau | chng | SendAbsUpdTime,
50
51      RecvMem_s | SendUpdMem_s | RecvAbsTime | a | chng | SendAbsUpdTime,
52      RecvMem_s | SendUpdMem_s | RecvAbsTime | b | chng | SendAbsUpdTime,
53      RecvMem_s | SendUpdMem_s | RecvAbsTime | ctau | chng | SendAbsUpdTime
54
55    },
56    comm({RecvAbsTime | RecvAbsTime -> RecvAbsTime,
57          SendAbsUpdTime | SendAbsUpdTime -> SendAbsUpdTime,
58          RecvMem_s | RecvMem_s -> RecvMem_s,
59          SendUpdMem_s | SendUpdMem_s -> SendUpdMem_s,
60          chng | chng -> chng},
61    (
62      sum t: Htime. sum t': Htime.
63    ( (lambda v1: Bool,
64              time':TimeSort,
65              time:TimeSort, predicate:Bool#TimeSort -> Bool .
66                (predicate(v1, time') && if(Urgent_G(a),
67                  forall t'':TimeSort . time <= t'' && t'' < time'
68                    => !(predicate(v1, t'')), true)))
69      (dc, pi_time(t'), pi_time(t),
70        (lambda v1:Bool, time:TimeSort. time >= 2))) ->
71          (RecvAbsTime(t) | a | chng({time}) | SendAbsUpdTime(t')@pi_time(t')))
72  ||
73    ( sum t: Htime. sum t': Htime. sum w1: Bool.
74    ( (lambda v1: Bool,
75              time':TimeSort,
```

```
76              time:TimeSort, predicate:Bool#TimeSort -> Bool .
77                (predicate(v1, time') && if(Urgent_G(b),
78                  forall t'':TimeSort . time <= t'' && t'' < time'
79                    => !(predicate(v1, t'')), true)))
80      (dc, pi_time(t'), pi_time(t),
81        (lambda v1:Bool, time:TimeSort. time <= 10 && time >= 1))
82        &&
83        (lambda v1: Bool, w1: Bool, time':TimeSort, time:TimeSort,
84                predicate:Bool#Bool#TimeSort -> Bool.(predicate(v1, w1, time')))
85          (dc, w1, pi_time(t'), pi_time(t),
86            (lambda v1: Bool, w1:Bool, time:TimeSort. w1 == false))) ->
87              (SendUpdMem_s(w1) | RecvAbsTime(t)
88              | b | chng({s, time}) | SendAbsUpdTime(t')@pi_time(t')))
89 ));
90
91 init hide({ctau},
92        allow({CommAbsTime | CommMem_s | a | chng | CommUpdMem_s | CommAbsUpdTime,
93                CommAbsTime | CommMem_s | b | chng | CommUpdMem_s | CommAbsUpdTime,
94                CommAbsTime | CommMem_s | ctau | chng |  CommUpdMem_s | CommAbsUpdTime,
95              },
96         comm({ SendAbsTime | RecvAbsTime -> CommAbsTime,
97                SendAbsUpdTime | RecvAbsUpdTime -> CommAbsUpdTime,
98                SendMem_s | RecvMem_s -> CommMem_s,
99                SendUpdMem_s | RecvUpdMem_s -> CommUpdMem_s
100             },
101        ProcMem(true) || ProcTime(htime(0, 0)) || ProcChi)));
```

B.2.4 Communication Example

```
1  sort ChiLabelsBasic = struct a | b | send_c | recv_c;
2
3  sort TimeSort = Nat;
4
5  sort Variables = struct s | time;
6
7  sort Htime = struct htime(pi_time: TimeSort, pi_counter: Nat);
8
9  act a, b;
10     send_c, recv_c, comm_c: Bool;
11 act ctau;
12 act SendAbsTime, RecvAbsTime, CommAbsTime,
13     SendAbsUpdTime, RecvAbsUpdTime, CommAbsUpdTime: Htime;
14     SendMem_s, RecvMem_s, CommMem_s: Bool;
15     SendUpdMem_s, RecvUpdMem_s, CommUpdMem_s: Bool;
16     chng: Set(Variables);
17
18 map Urgent: ChiLabelsBasic -> Bool;
19 eqn Urgent(a) = true;
20     Urgent(b) = false;
21     Urgent(send_c) = true;
22     Urgent(recv_c) = false;
23
24 proc ProcTime(t: Htime) = sum t': Htime. pred_htime(t, t')
25                          -> SendAbsTime(t) | RecvAbsUpdTime(t'). ProcTime(t');
26
27 map pred_htime: Htime#Htime -> Bool;
28 var t, t': Htime;
29 eqn pred_htime(t, t') = (t' == htime(pi_time(t), pi_counter(t)+1)) ||
30                        (pi_time(t') > pi_time(t) && pi_counter(t') == 0);
31
32 proc ProcMems: Bool) =
33     sum s': Bool . SendMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
34   +                SendMem_s(s) | CommUpdMem_s(s ) . ProcMem(s )
35   + sum s': Bool . CommMem_s(s) | RecvUpdMem_s(s') . ProcMem(s')
```

```
36    +                 CommMem_s(s) | CommUpdMem_s(s ) . ProcMem(s );
37
38  glob dc: Bool;
39  proc ProcChi =
40    allow(
41      {
42        RecvMem_s                  | RecvAbsTime | a | chng | SendAbsUpdTime,
43        RecvMem_s | SendUpdMem_s | RecvAbsTime | a | chng | SendAbsUpdTime,
44                    SendUpdMem_s | RecvAbsTime | a | chng | SendAbsUpdTime,
45                                   RecvAbsTime | a | chng | SendAbsUpdTime,
46
47        RecvMem_s                  | RecvAbsTime | b | chng | SendAbsUpdTime,
48        RecvMem_s | SendUpdMem_s | RecvAbsTime | b | chng | SendAbsUpdTime,
49                    SendUpdMem_s | RecvAbsTime | b | chng | SendAbsUpdTime,
50                                   RecvAbsTime | b | chng | SendAbsUpdTime,
51
52        RecvMem_s                  | RecvAbsTime | b | ctau | SendAbsUpdTime,
53        RecvMem_s | SendUpdMem_s | RecvAbsTime | b | ctau | SendAbsUpdTime,
54                    SendUpdMem_s | RecvAbsTime | b | ctau | SendAbsUpdTime,
55                                   RecvAbsTime | b | ctau | SendAbsUpdTime,
56
57        RecvMem_s                  | RecvAbsTime | send_c | chng | SendAbsUpdTime,
58        RecvMem_s | SendUpdMem_s | RecvAbsTime | send_c | chng | SendAbsUpdTime,
59                    SendUpdMem_s | RecvAbsTime | send_c | chng | SendAbsUpdTime,
60                                   RecvAbsTime | send_c | chng | SendAbsUpdTime,
61
62        RecvMem_s                  | RecvAbsTime | recv_c | chng | SendAbsUpdTime,
63        RecvMem_s | SendUpdMem_s | RecvAbsTime | recv_c | chng | SendAbsUpdTime,
64                    SendUpdMem_s | RecvAbsTime | recv_c | chng | SendAbsUpdTime,
65                                   RecvAbsTime | recv_c | chng | SendAbsUpdTime,
66
67        RecvMem_s                  | RecvAbsTime | comm_c | chng | SendAbsUpdTime,
68        RecvMem_s | SendUpdMem_s | RecvAbsTime | comm_c | chng | SendAbsUpdTime,
69                    SendUpdMem_s | RecvAbsTime | comm_c | chng | SendAbsUpdTime,
70                                   RecvAbsTime | comm_c | chng | SendAbsUpdTime,
71
72        RecvMem_s                  | RecvAbsTime | comm_c | chng | chng | SendAbsUpdTime,
73        RecvMem_s | SendUpdMem_s | RecvAbsTime | comm_c | chng | chng | SendAbsUpdTime,
74                    SendUpdMem_s | RecvAbsTime | comm_c | chng | chng | SendAbsUpdTime,
75                                   RecvAbsTime | comm_c | chng | chng | SendAbsUpdTime,
76
77        RecvMem_s                  | RecvAbsTime | ctau | chng | SendAbsUpdTime,
78        RecvMem_s | SendUpdMem_s | RecvAbsTime | ctau | chng | SendAbsUpdTime,
79                    SendUpdMem_s | RecvAbsTime | ctau | chng | SendAbsUpdTime,
80                                   RecvAbsTime | ctau | chng | SendAbsUpdTime
81      },
82    comm( {RecvAbsTime | RecvAbsTime -> RecvAbsTime,
83           SendAbsUpdTime | SendAbsUpdTime -> SendAbsUpdTime,
84           RecvMem_s | RecvMem_s -> RecvMem_s,
85           SendUpdMem_s | SendUpdMem_s -> SendUpdMem_s,
86           send_c | recv_c -> comm_c,
87           chng | chng -> chng},
88            ( sum t: Htime. sum t': Htime. sum v1: Bool.
89            ( (lambda v1: Bool, time': TimeSort, time: TimeSort,
90                predicate: Bool#TimeSort -> Bool .
91                  (predicate(v1, time') && if(Urgent(send_c),
92                    forall x: TimeSort .
93                      time <= x && x < time' => !(predicate(v1, x)), true)))
94                      (dc, pi_time(t'), pi_time(t),
95                      (lambda v1: Bool, time: TimeSort. time >= 2))) ->
96              ( RecvMem_s(v1) | RecvAbsTime(t) | send_c(!v1) | chng({time})
97              | SendAbsUpdTime(t') @pi_time(t')
98              ))
99            ||
100             ( sum t: Htime. sum t': Htime. sum w1: Bool.
```

```
101             ( (lambda v1: Bool, time': TimeSort, time: TimeSort,
102                predicate: Bool#TimeSort -> Bool .
103                  (predicate(v1, time') && if(Urgent(recv_c),
104                    forall x: TimeSort .
105                      time <= x && x < time' => !(predicate(v1, x)), true)))
106                      (dc, pi_time(t'), pi_time(t),
107                      (lambda v1: Bool, time: TimeSort. time >= 2))) ->
108             ( SendUpdMem_s(w1) | RecvAbsTime(t)
109             | recv_c(w1) | chng({s, time})
110             | SendAbsUpdTime(t') @pi_time(t')
111             ))));
112
113 init hide({ctau},
114      allow({CommAbsTime | CommMem_s | a | chng | CommUpdMem_s | CommAbsUpdTime,
115             CommAbsTime | CommMem_s | b | chng | CommUpdMem_s | CommAbsUpdTime,
116             CommAbsTime | CommMem_s | ctau | chng | CommUpdMem_s | CommAbsUpdTime,
117             CommAbsTime | CommMem_s | ctau | chng | chng | CommUpdMem_s | CommAbsUpdTime,
118             CommAbsTime | CommMem_s | CommUpdMem_s | chng |chng |comm_c | CommAbsUpdTime,
119             CommAbsTime | CommMem_s | CommUpdMem_s | chng |comm_c | CommAbsUpdTime
120           },
121      comm({ SendAbsTime | RecvAbsTime -> CommAbsTime,
122             SendAbsUpdTime | RecvAbsUpdTime -> CommAbsUpdTime,
123             SendMem_s | RecvMem_s -> CommMem_s,
124             SendUpdMem_s | RecvUpdMem_s -> CommUpdMem_s
125           },
126      ProcMem(true) || ProcTime(htime(0, 0)) || ProcChi)));
```

B.3 The Wafer Dryer Facility Model

This section presents the original mCRL2 source code model for the wafer dryer facility from Chapter 6. The model that has been used to generate traces is a derivative from the one stated here. In the derivative model the dense time domain has been replaced by a model of ticks, with a resolution of one tick per second. The progress of time is modeled as a data parameter, and functions that ensure that the value of the time always increases. The derivative model is not included.

```
1  sort Places     = struct S0 | S1 | S2 | S3 | S4;
2       Wafer      = struct wafer(Id: Nat, Place: Places,
3                                 State: Bool, Stamp: Real);
4
5  map  getWafersID: Set(Wafer) -> Set(Nat);
6  var  sw: Set(Wafer);
7  eqn  getWafersID(sw)={i: Nat | exists x: Wafer. x in sw && Id(x) == i && i < 6};
8
9  map  getWafersPlaces: Set(Wafer) -> Set(Places);
10 var  sw: Set(Wafer);
11 eqn  getWafersPlaces(sw) = {i: Places |
12                                exists x: Wafer. x in sw && Place(x) == i };
13
14 map  frac: Real -> Real;
15 var  x: Real;
16 eqn  frac(x) = x - floor(x);
17
18 map  flip: Bool -> Bool;
19 var  x: Bool;
20 eqn  flip(x) = !x;
21
22 map rate, time2turn, time2move, time2insert, time2remove, drytime: Nat;
23     offset: Real;
24 eqn rate        = 30; % Rate at which wafers enter the system
25     time2turn   =  5; % Time required to turn wafers
```

```
26      time2move   =  3; % Time required to move a wafer
27      time2insert =  3; % Time required to insert a wafer into the system
28      time2remove =  3; % Time required to exit a wafer from the system
29      drytime     = 60; % Required time to dry a wafer
30      offset      =  1; % Offset is required to escape "action@0"
31  act
32    S0toS1_begin, S0toS3_begin, S1toS0_begin, S1toS4_begin,
33    S1toS2_begin, S2toS1_begin, S2toS3_begin, S3toS2_begin,
34    S3toS0_begin, S3toS4_begin,
35    S0toS1_end, S0toS3_end, S1toS0_end, S1toS2_end,
36    S1toS4_end, S2toS1_end, S2toS3_end, S3toS2_end,
37    S3toS0_end, S3toS4_end: Nat;
38    S0toS1_begin_dc, S0toS3_begin_dc, S1toS0_begin_dc, S1toS4_begin_dc,
39    S1toS2_begin_dc, S2toS1_begin_dc, S2toS3_begin_dc, S3toS2_begin_dc,
40    S3toS0_begin_dc, S3toS4_begin_dc,
41    S0toS1_end_dc, S0toS3_end_dc, S1toS0_end_dc, S1toS2_end_dc,
42    S1toS4_end_dc, S2toS1_end_dc, S2toS3_end_dc, S3toS2_end_dc,
43    S3toS0_end_dc, S3toS4_end_dc: Nat;
44    S0toS1_begin_c, S0toS3_begin_c, S1toS0_begin_c, S1toS4_begin_c,
45    S1toS2_begin_c, S2toS1_begin_c, S2toS3_begin_c, S3toS2_begin_c,
46    S3toS0_begin_c, S3toS4_begin_c,
47    S0toS1_end_c, S0toS3_end_c, S1toS0_end_c, S1toS2_end_c,
48    S1toS4_end_c, S2toS1_end_c, S2toS3_end_c, S3toS2_end_c,
49    S3toS0_end_c, S3toS4_end_c: Nat;
50    skip;
51
52  %%%%%%%%%%%%%%%%%%%%
53  %% Dryer System %%
54  %%%%%%%%%%%%%%%%%%%%
55  proc DS(S1: Bool, S2: Bool, S3: Bool) =
56           sum n: Nat.  sum t: Real. (!S1) ->
57                           S0toS1_begin_dc(n)@t
58                         . S0toS1_end_dc(n)@(t+time2insert)
59                         . DS(true, S2, S3)
60         + sum n: Nat.  sum t: Real. (!S3) ->
61                           S0toS3_begin_dc(n)@t
62                         . S0toS3_end_dc(n)@(t+time2insert)
63                         . DS(S1, S2, true)
64         + sum n: Nat.  sum t: Real.
65                           S1toS0_begin_dc(n)@t
66                         . S1toS0_end_dc(n)@(t+time2remove)
67                         . DS(false, S2, S3)
68         + sum n: Nat.  sum t: Real.
69                           S1toS4_begin_dc(n)@t
70                         . S1toS4_end_dc(n)@(t+time2remove)
71                         . DS(false, S2, S3)
72         + sum n, m: Nat.sum t: Real.
73                           S1toS2_begin_dc(n) | S2toS1_begin_dc(m)@t
74                         . S1toS2_end_dc(n)   | S2toS1_end_dc(m)@(t+time2turn)
75                         . DS(S2, S1, S3)
76         + sum n: Nat.  sum t: Real. (!S3) ->
77                           S2toS3_begin_dc(n)@t
78                         . S2toS3_end_dc(n)@(t+time2move)
79                         . DS(S1, false, S2)
80         + sum n: Nat.  sum t: Real. (!S2) ->
81                           S3toS2_begin_dc(n)@t
82                         . S3toS2_end_dc(n)@(t+time2move)
83                         . DS(S1, S3, false)
84         + sum n: Nat.  sum t: Real.
85                           S3toS4_begin_dc(n)@t
86                         . S3toS4_end_dc(n)@(t+time2move)
87                         . DS(S1, S2, false)
88         + sum n: Nat.  sum t: Real.
89                           S3toS0_begin_dc(n)@t
90                         . S3toS0_end_dc(n)@(t+time2move)
```

```
91                          . DS(S1, S2, false);
92
93  %%%%%%%%%%%%%%%%%
94  %% Controller %%
95  %%%%%%%%%%%%%%%%%
96  proc C(wafers: Set(Wafer), time: Real) =
97          % New wafer enters the dryer facility
98            sum n: Nat.     sum t: Real. ((frac((time-offset)/ rate) == 0)
99                            && !(n in getWafersID(wafers)))
100                           -> S0toS1_begin_c(n)@time
101                             . S0toS1_end_c(n)@t
102                             . C({wafer(n, S1, true, time)} + wafers, t)
103         + sum n: Nat.     sum t: Real. ((frac((time-offset)/ rate) == 0)
104                           && !(n in getWafersID(wafers)))
105                           -> S0toS3_begin_c(n)@time
106                             . S0toS3_end_c(n)@t
107                             . C({wafer(n, S3, true, time)} + wafers, t)
108         % Move wafer inside dryer facility
109         + sum w: Wafer.   sum t: Real. ((frac((time-offset)/ rate) != 0)
110                           && (frac((t-offset)/ rate)!=0)
111                           && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
112                           && (Place(w) == S2)
113                           && !(S3 in getWafersPlaces(wafers)))
114                           -> S2toS3_begin_c(Id(w))@time
115                             . S2toS3_begin_c(Id(w))@t
116                             . C((wafers - {w})
117                                 + {wafer(Id(w), S3, State(w), Stamp(w))}, t)
118         + sum w: Wafer.   sum t: Real. ((frac((time-offset)/ rate) != 0)
119                           && (frac((t-offset)/ rate)!=0)
120                           && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
121                           && (Place(w) == S3)
122                           && !(S2 in getWafersPlaces(wafers)))
123                           -> S3toS2_begin_c(Id(w))@time
124                            . S3toS2_begin_c(Id(w))@t
125                            . C((wafers - {w})
126                               + {wafer(Id(w), S2, State(w), Stamp(w))}, t)
127         % Turn wafer
128         + sum w, x: Wafer.sum t: Real. ((frac((time-offset)/ rate) != 0)
129                           && (frac((t-offset)/ rate)!=0)
130                           && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
131                           && (Place(w) == S1) && (Place(x) == S2))
132                           -> S1toS2_begin_c(Id(w)) | S2toS1_begin_c(Id(x))@time
133                             . S1toS2_end_c(Id(w)) | S2toS1_end_c(Id(x))@t
134                             . C((wafers - {w, x})
135                                 + {wafer(Id(w), S2, flip(State(w)), Stamp(w)),
136                                    wafer(Id(x), S1, flip(State(x)), Stamp(x))}, t)
137         + sum w: Wafer.   sum t: Real. ((frac((time-offset)/ rate) != 0)
138                           && (frac((t-offset)/ rate)!=0)
139                           && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
140                           && (Place(w) == S1)
141                           && !(S2 in getWafersPlaces(wafers)))
142                           -> S1toS2_begin_c(Id(w)) | S2toS1_begin_c(0)@time
143                             . S1toS2_end_c(Id(w)) | S2toS1_end_c(0)@t
144                             . C((wafers-{w})
145                                 + {wafer(Id(w), S2, flip(State(w)), Stamp(w))}, t)
146         + sum w: Wafer.   sum t: Real. ((frac((time-offset)/ rate) != 0)
147                           && (frac((t-offset)/ rate)!=0)
148                           && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
149                           && (Place(w) == S2)
150                           && !(S1 in getWafersPlaces(wafers)))
151                           -> S2toS1_begin_c(0) | S2toS1_begin_c(Id(w))@time
152                             . S1toS2_end_c(0) | S2toS1_end_c(Id(w))@t
153                             . C((wafers-{w}) +
154                                   {wafer(Id(w), S1, flip(State(w)), Stamp(w))}, t)
155         % Wafer departure
```

```
156         + sum w: Wafer.  sum t: Real. ((frac((time-offset)/ rate) != 0)
157                          && (frac((t-offset)/ rate)!=0)
158                          && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
159                          && (Place(w) == S1)
160                          && (t - (drytime + Stamp(w)) >=0))
161                          -> S1toS4_begin_c(Id(w))@time
162                           . S1toS4_end_c(Id(w))@t
163                           . C(wafers-{w}, t)
164         + sum w: Wafer.  sum t: Real. ((frac((time-offset)/ rate) != 0)
165                          && (frac((t-offset)/ rate)!=0)
166                          && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
167                          && (Place(w) == S3)
168                          && (t - (drytime + Stamp(w)) >=0))
169                          -> S3toS4_begin_c(Id(w))@time
170                           . S3toS4_end_c(Id(w))@t
171                           . C(wafers-{w}, t)
172         + sum w: Wafer.  sum t: Real. ((frac((time-offset)/ rate) != 0)
173                          && (frac((t-offset)/ rate)!=0)
174                          && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
175                          && (Place(w) == S1)
176                          && (t - (drytime + Stamp(w)) >=0)
177                          && (State(w) == false))
178                          -> S1toS0_begin_c(Id(w))@time
179                           . S1toS0_end_c(Id(w))@t
180                           . C(wafers-{w}, t)
181         + sum w: Wafer.  sum t: Real. ((frac((time-offset)/ rate) != 0)
182                          && (frac((t-offset)/ rate)!=0)
183                          && (frac((time-offset)/ rate) < frac((t-offset)/ rate))
184                          && (Place(w) == S3)
185                          && (t - (drytime + Stamp(w)) >=0)
186                          && (State(w) == false))
187                          -> S3toS0_begin_c(Id(w))@time
188                           . S3toS0_end_c(Id(w))@t
189                           . C(wafers-{w}, t)
190         % time expires
191         + sum t: Real.      ((frac((time-offset)/ rate) != 0)
192                          && (frac((t-offset)/ rate)!=0)
193                          && (frac((time-offset)/ rate) < frac((t-offset)/ rate)))
194                          -> skip
195                           . C(wafers, t);
196
197 init
198 allow({
199   S0toS1_begin, S0toS1_end,
200   S0toS3_begin, S0toS3_end,
201   S1toS0_begin, S1toS0_end,
202   S1toS4_begin, S1toS2_end,
203   S1toS2_begin, S1toS4_end,
204   S2toS1_begin, S2toS1_end,
205   S2toS3_begin, S2toS3_end,
206   S3toS2_begin, S3toS2_end,
207   S3toS0_begin, S3toS0_end,
208   S3toS4_begin, S3toS4_end,
209   skip},
210   comm({
211     S0toS1_begin_dc | S0toS1_begin_c  ->  S0toS1_begin,
212     S0toS3_begin_dc | S0toS3_begin_c  ->  S0toS3_begin,
213     S1toS0_begin_dc | S1toS0_begin_c  ->  S1toS0_begin,
214     S1toS4_begin_dc | S1toS4_begin_c  ->  S1toS4_begin,
215     S1toS2_begin_dc | S1toS2_begin_c  ->  S1toS2_begin,
216     S2toS1_begin_dc | S2toS1_begin_c  ->  S2toS1_begin,
217     S2toS3_begin_dc | S2toS3_begin_c  ->  S2toS3_begin,
218     S3toS2_begin_dc | S3toS2_begin_c  ->  S3toS2_begin,
219     S3toS0_begin_dc | S3toS0_begin_c  ->  S3toS0_begin,
220     S3toS4_begin_dc | S3toS4_begin_c  ->  S3toS4_begin,
```

```
221      S0toS1_end_dc   | S0toS1_end_c    -> S0toS1_end,
222      S0toS3_end_dc   | S0toS3_end_c    -> S0toS3_end,
223      S1toS0_end_dc   | S1toS0_end_c    -> S1toS0_end,
224      S1toS2_end_dc   | S1toS2_end_c    -> S1toS2_end,
225      S1toS4_end_dc   | S1toS4_end_c    -> S1toS4_end,
226      S2toS1_end_dc   | S2toS1_end_c    -> S2toS1_end,
227      S2toS3_end_dc   | S2toS3_end_c    -> S2toS3_end,
228      S3toS2_end_dc   | S3toS2_end_c    -> S3toS2_end,
229      S3toS0_end_dc   | S3toS0_end_c    -> S3toS0_end,
230      S3toS4_end_dc   | S3toS4_end_c    -> S3toS4_end},
231        C({}, offset) || DS(false, false, false)
232     )
233   );
```

B.4 Minimal Process Theory Models

This section presents the original mCRL2 source code models that have been created using the semantic approach for the MPT example (including predicates) from Chapter 6. The semantics is modeled in:

```
1  % This mCRL2 model describes the implementation of the Structural Operational
2  % Semantic deduction rules mentioned in:
3  %
4  %   J.C.M. Baeten, T. Basten, and M.A. Reniers
5  %   Process Algebra: Equational Theories of Communicating Processes
6  %   (Cambridge Tracts in Theoretical Computer Science)
7  %
8  % The locations of the corresponding deduction rules are mentioned above them.
9
10 % Set of Action/Predicate labels
11 sort AL = struct a0 | a1 | a2 | term;
12
13 % Signature
14 sort T = struct zero?is_zero
15                | one?is_one
16                | a0(pi_1: T)?is_a0
17                | a1(pi_1: T)?is_a1
18                | a2(pi_1: T)?is_a2
19                | alt(pi_1: T, pi_2: T)?is_alt
20                | par(pi_1: T, pi_2: T)?is_par
21                | seq(pi_1: T, pi_2: T)?is_seq
22                ;
23
24 sort Solution = struct sol(pi_l: AL, pi_t: T);
25
26 map R, R_a0, R_a1, R_a2, R_alt_1, R_alt_2: T -> Set(Solution);
27     R_par_0, R_par_1, R_par_2: T -> Set(Solution);
28     R_t1, R_t2, R_t3, R_seq_1, R_seq_2, R_seq_3: T -> Set(Solution);
29     TR: T -> Set(Solution);
30 var p: T;
31 eqn R(p) = R_a0(p) + R_a1(p) + R_a2(p) + R_alt_1(p) + R_alt_2(p)
32          + R_par_1(p) + R_par_2(p) + R_seq_2(p) + R_seq_3(p);
33     TR(p) = R_par_0(p) + R_t1(p) + R_t2(p) + R_t3(p) + R_seq_1(p);
34
35 map sigma_a0, sigma_a1, sigma_a2, sigma_alt_1: T -> Bool;
36     sigma_alt_2, sigma_par_1, sigma_par_2: T -> Bool;
37     Cond_a0, Cond_a1, Cond_a2, Cond_alt_1, Cond_alt_2: List(AL)#AL-> Bool;
38     Cond_par_1, Cond_par_2: List(AL)#AL-> Bool;
39     mu_a0_x0, mu_a1_x0, mu_a2_x0, mu_alt_1_x0, mu_alt_2_x0: T -> T;
40     mu_par_1_y0, mu_par_1_x1, mu_par_2_x0, mu_par_2_y1: T -> T;
41 var l: AL;
```

```
42      ls: List(AL);
43      p: T;
44
45      % Page 74, Rule 1
46  eqn Cond_a0(ls, l) = l == a0;
47      sigma_a0(p) = true;
48      mu_a0_x0(p) = p;
49      R_a0(p)     = {t: Solution | is_a0(p)
50                                 && sigma_a0(pi_t(t))
51                                 && Cond_a0([], pi_l(t))
52                                 && mu_a0_x0(pi_t(t)) == pi_1(p)};
53
54      % Page 74, Rule 1
55      Cond_a1(ls, l) = l == a1;
56      sigma_a1(p) = true;
57      mu_a1_x0(p) = p;
58      R_a1(p)     = {t: Solution | is_a1(p)
59                                 && sigma_a1(pi_t(t))
60                                 && Cond_a1([], pi_l(t))
61                                 && mu_a1_x0(pi_t(t)) == pi_1(p)};
62
63      % Page 74, Rule 1
64      Cond_a2(ls, l) = l == a2;
65      sigma_a2(p) = true;
66      mu_a2_x0(p) = p;
67      R_a2(p)     = {t: Solution | is_a2(p)
68                                 && sigma_a2(pi_t(t))
69                                 && Cond_a2([], pi_l(t))
70                                 && mu_a2_x0(pi_t(t)) == pi_1(p)};
71
72      % Page 74, Rule 2
73      sigma_alt_1(p) = true;
74      mu_alt_1_x0(p) = p;
75      R_alt_1(p)     = {t: Solution | is_alt(p)
76                                    && sigma_alt_1(pi_t(t))
77                                    && sol(pi_l(t), mu_alt_1_x0(pi_t(t))) in R(pi_1(p))};
78
79      % Page 74, Rule 3
80      sigma_alt_2(p) = true;
81      mu_alt_2_x0(p) = p;
82      R_alt_2(p)     = {t: Solution | is_alt(p)
83                                    && sigma_alt_2(pi_t(t))
84                                    && sol(pi_l(t), mu_alt_2_x0(pi_t(t))) in R(pi_2(p))};
85
86      % Page 84, Rule 1
87      R_t1(p)     = {t: Solution | is_one(p) && pi_l(t) == term && (pi_t(t)) == p};
88
89      % Page 84, Rule 2
90      R_t2(p)     = {t: Solution | is_alt(p)
91                                 && p == pi_t(t)
92                                 && term == pi_l(t)
93                                 && sol(term, pi_1(p)) in TR(pi_1(p))};
94      % Page 84, Rule 3
95      R_t3(p)     = {t: Solution | is_alt(p)
96                                 && p == pi_t(t)
97                                 && term == pi_l(t)
98                                 && sol(term, pi_2(p)) in TR(pi_2(p))};
99
100     % Page 175, Rule 1
101     R_seq_1(p)  = {t: Solution | is_seq(p)
102                                && sol(pi_l(t), pi_1(pi_t(t))) in R(pi_1(p))
103                                && sol(pi_l(t), pi_2(pi_t(t))) in R(pi_2(p))
104                                && pi_l(t) == term
105                                && is_seq(pi_t(t))};
106
```

```
107     % Page 175, Rule 2
108     R_seq_2(p)  = {t: Solution | is_seq(p)
109                                && sol(pi_l(t), pi_1(pi_t(t))) in R(pi_1(p))
110                                && pi_2(pi_t(t)) == pi_2(p)
111                                && is_seq(pi_t(t))};
112
113     % Page 175, Rule 3
114     R_seq_3(p)  = {t: Solution | is_seq(p)
115                                && sol(term, pi_1(p)) in TR(pi_1(p))
116                                && t in R(pi_2(p))};
117
118     % Page 216, Rule 1
119     R_par_0(p)   = {t: Solution | is_par(p)
120                                 && p == pi_t(t)
121                                 && sol(term, pi_1(p)) in TR(pi_1(p))
122                                 && sol(term, pi_2(p)) in TR(pi_2(p))
123                                 && pi_l(t) == term};
124
125     % Page 216, Rule 3
126     sigma_par_1(p) = is_par(p);
127     mu_par_1_y0(p) = pi_1(p);
128     mu_par_1_x1(p) = pi_2(p);
129     R_par_1(p)     = {t: Solution | is_par(p)
130                                   && sigma_par_1(pi_t(t))
131                                   && sol(pi_l(t), mu_par_1_y0(pi_t(t))) in R(pi_1(p))
132                                   && mu_par_1_x1(pi_t(t)) == pi_2(p)};
133
134     % Page 216, Rule 4
135     sigma_par_2(p) = is_par(p) ;
136     mu_par_2_x0(p) = pi_1(p);
137     mu_par_2_y1(p) = pi_2(p);
138     R_par_2(p)     = {t: Solution | is_par(p)
139                                   && sigma_par_2(pi_t(t))
140                                   && mu_par_2_x0(pi_t(t)) == pi_1(p)
141                                   && sol(pi_l(t), mu_par_2_y1(pi_t(t))) in R(pi_2(p))};
142
143 act tr: AL;
144     pr: AL;
145
146 proc X(p: T) = sum s: Solution. (s in  R(p)) -> tr(pi_l(s)) . X(pi_t(s))
147             + sum s: Solution. (s in TR(p)) -> pr(pi_l(s)) . X(pi_t(s));
```

The different input models that have been used to generate the LTSs are described in:

```
1  % Figure 8.1a
2  init X(a0(a1(a2(zero))));
3
4  % Figure 8.1b
5  init X(par(a1(zero)), a2(zero)));
6
7  % Figure 8.1c
8  init X(alt(a0(zero), a1(a2(zero))));
9
10 % Figure 8.2
11 init X(alt(a0(zero), a1(one)));
```

B.5 Semantically Engineered mCRL2 Models

B.5.1 Language Semantics

The language semantics describes the implementation of the semantics that hold for all untimed mCRL2 models. To create a valid LPS, this semantics is extended with the model specific (static) semantics (Appendix B.5.2) and a specific model (Appendix B.5.3).

```
%%%%%%%%%%%%
%% Sorts %%
%%%%%%%%%%%%

% An argument of a valuation.
sort Argument = struct argument(variable: Variable, valvalue: Value);

% Valuation.
sort Valuation = List(Argument);

% Data expression.
sort DataExpression = struct
                        de_var(dvr: Variable)?is_de_var
                      | de_val(dvl: Value)?is_de_val
                      | de_expr_1(f: Func, expr_1: DataExpression)?is_de_expr_1
                      | de_expr_2(f: Func, expr_1: DataExpression,
                                           expr_2: DataExpression)?is_de_expr_2;
% Syntactic action.
sort ActionSyntax  =
       struct Act(ActionLabel: ActionLabel, args: List(DataExpression))
            | ActionTau;

% The sort to model process parameters
sort PP  = struct pp(variable: Variable, dataexpression: DataExpression);

% The sort used to model communication
sort Communication =
       struct communication(CmI: List(ActionLabel), CmR: ActionLabel);

% The signature of an mCRL2 process term
sort ProcessTerm = struct
                    Checkmark?is_Checkmark
                  | Deadlock?is_Deadlock
                  | Alpha(pi_multiaction: List(ActionSyntax))?is_Alpha
                  | Alt(pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Alt
                  | Seq(pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Seq
                  | Cond1(pi_C: DataExpression, pi_1: ProcessTerm)?is_Cond1
                  | Cond2(pi_C: DataExpression,
                      pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Cond2
                  | Sum(pi_v: Variable, pi_1: ProcessTerm)?is_Sum
                  | Par(pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Par
                  | Lmerge(pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Lmerge
                  | Sync(pi_1: ProcessTerm, pi_2: ProcessTerm)?is_Sync
                  | Allow(pi_V: Set(Bag(ActionLabel)), pi_1: ProcessTerm)?is_Allow
                  | Block(pi_B: Set(ActionLabel), pi_1: ProcessTerm)?is_Block
                  | Rename(pi_Ren: ActionLabel -> ActionLabel,
                      pi_1: ProcessTerm)?is_Rename
                  | Hide(pi_I: Set(ActionLabel), pi_1: ProcessTerm)?is_Hide
                  | Prehide(pi_U: Set(ActionLabel), pi_1: ProcessTerm)?is_Prehide
                  | Comm(pi_CL: List(Communication), pi_1: ProcessTerm)?is_Comm
                  | Def(pi_P: ProcessLabel, ppl: List(PP))?is_Def;

% Semantic action
sort ActionSemantic =
```

```
55          struct ActSem(ActionLabel: ActionLabel, args: List(Value));
56
57  % The sort for an action transition
58  sort ActionTransition =
59          struct at(pi_ac: List(ActionSemantic),
60                    pi_t: ProcessTerm,
61                    pi_sigma': Valuation);
62
63  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
64  %% Mappings to model the deduction rules %%
65  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
66
67  % The solution functions that compute the transition relations.
68  map R,
69      R_Alpha,
70      R_Alt_1, R_Alt_2, R_Alt_3, R_Alt_4,
71      R_Seq_1, R_Seq_2,
72      R_Cond1_1, R_Cond1_2,
73      R_Cond2_1, R_Cond2_2, R_Cond2_3, R_Cond2_4,
74      R_Sum_1, R_Sum_2,
75      R_Par_1, R_Par_2, R_Par_3, R_Par_4, R_Par_5, R_Par_6, R_Par_7, R_Par_8,
76      R_Sync_1, R_Sync_2, R_Sync_3, R_Sync_4,
77      R_Lmerge_1, R_Lmerge_2,
78      R_Allow_1, R_Allow_2,
79      is_Block_1, is_Block_2,
80      R_Rename_1, R_Rename_2,
81      R_Hide_1, R_Hide_2,
82      R_Prehide_1, R_Prehide_2,
83      R_Comm_1, R_Comm_2,
84      R_Def_1, R_Def_2
85      : ProcessTerm#Valuation -> Set(ActionTransition);
86  var p: ProcessTerm;
87      s: Valuation;
88  eqn R(p,s)    = R_Alpha(p,s)
89                + R_Alt_1(p,s) + R_Alt_2(p,s) + R_Alt_3(p,s) + R_Alt_4(p,s)
90                + R_Seq_1(p,s) + R_Seq_2(p,s)
91                + R_Cond1_1(p,s) + R_Cond1_2(p,s)
92                + R_Cond2_1(p,s) + R_Cond2_2(p,s) + R_Cond2_3(p,s) + R_Cond2_4(p,s)
93                + R_Sum_1(p,s)  + R_Sum_2(p,s)
94                + R_Par_1(p,s) + R_Par_2(p,s) + R_Par_3(p,s) + R_Par_4(p,s)
95                + R_Par_5(p,s)
96                + R_Par_6(p,s) + R_Par_7(p,s) + R_Par_8(p,s)
97                + R_Sync_1(p,s) + R_Sync_2(p,s) + R_Sync_3(p,s) + R_Sync_4(p,s)
98                + R_Lmerge_1(p,s) + R_Lmerge_2(p,s)
99                + R_Allow_1(p,s) + R_Allow_2(p,s)
100               + is_Block_1(p,s) + is_Block_2(p,s)
101               + R_Rename_1(p,s) + R_Rename_2(p,s)
102               + R_Hide_1(p,s) + R_Hide_2(p,s)
103               + R_Prehide_1(p,s) + R_Prehide_2(p,s)
104               + R_Comm_1(p,s) + R_Comm_2(p,s)
105               + R_Def_1(p,s)  + R_Def_2(p,s);
106
107     %%
108     %% ------------------------------------------
109     %% (Alpha,s) --[[Alpha]](s)--> Checkmark
110     R_Alpha(p,s) = if(is_Alpha(p),
111       { r: ActionTransition |
112               pi_ac(r)      == Sem_ActList(pi_multiaction(p), s)
113            && is_Checkmark(pi_t(r))
114            && pi_sigma'(r) == s},
115       {});
116
117     % (p,s) --m--> Checkmark
118     % ------------------------------
119     % (p + q,s) --m--> Checkmark
```

```
120     R_Alt_1(p,s) = if(is_Alt(p), { r: ActionTransition |
121                         r in R(pi_1(p),s)
122                     && is_Checkmark((pi_t(r)))
123                     && pi_sigma'(r) == s}, {});
124
125     % (p,s) --m-- -> (p',s')
126     % ----------------------
127     % (p + q, s) --m-- -> (p',s')
128     R_Alt_2(p,s) = if(is_Alt(p), { r: ActionTransition |
129                         r in R(pi_1(p),s)
130                     && !is_Checkmark(pi_t(r))}, {});
131
132     % (q,s) --m--> Checkmark
133     % ----------------------
134     % (p + q,s) --m--> Checkmark
135     R_Alt_3(p,s) = if(is_Alt(p), { r: ActionTransition |
136                         r in R(pi_2(p),s)
137                     && is_Checkmark(pi_t(r))
138                     && pi_sigma'(r) == s}, {});
139
140     % (p,s) --m-- -> (q',s')
141     % ----------------------
142     % (p + q, s) --m-- -> (q',s')
143     R_Alt_4(p,s) = if(is_Alt(p), { r: ActionTransition |
144                         r in R(pi_2(p),s)
145                     && !is_Checkmark(pi_t(r))}, {});
146
147     % (p,s) --m--> Checkmark
148     % ----------------------
149     % (p . q,s) --m-- -> (q,s)
150     R_Seq_1(p,s) = if(is_Seq(p), { r: ActionTransition |
151                         at(pi_ac(r), Checkmark, pi_sigma'(r)) in R(pi_1(p),s)
152                     && pi_t(r) == pi_2(p)
153                     && pi_sigma'(r) == s}, {});
154
155     %   (p,s) --m-- -> (p',s')
156     % ----------------------
157     % (p . q,s) --m-- -> (p'. q, s')
158     R_Seq_2(p,s) = if(is_Seq(p), { r: ActionTransition |
159                         is_Seq(pi_t(r))
160                     && at(pi_ac(r), pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p),s)
161                     && pi_2(pi_t(r)) == pi_2(p)
162                     && !is_Checkmark(pi_1(pi_t(r)))}, {});
163
164     % (p,s) --m--> Checkmark && [[b]](s)==true
165     % ----------------------------------------
166     % (b -> p,s) --m--> Checkmark
167     R_Cond1_1(p,s)= if(is_Cond1(p)
168        && Cast2InternalBool(Sem_Dex(pi_C(p), s)),
169                   { r: ActionTransition |
170                         r in R(pi_1(p), s)
171                     && is_Checkmark(pi_t(r))
172                     && pi_sigma'(r) == s, {});
173
174     % (p,s) --m-- -> (p',s') && [[b]](s) == true
175     % ----------------------------------------
176     % (b -> p,s) --m-- -> (p',s')
177     R_Cond1_2(p,s)= if(is_Cond1(p)
178         && Cast2InternalBool(Sem_Dex(pi_C(p), s)),
179                  { r: ActionTransition |
180                         r in R(pi_1(p), s)
181                     && !is_Checkmark(pi_t(r))}, {});
182
183     % (p,s) --m--> Checkmark && [[b]](s) == true
184     % ------------------------------------------
```

```
% (b -> p <> q,s) —m—> Checkmark
R_Cond2_1(p,s)= if(is_Cond2(p)
    && Cast2InternalBool(Sem_Dex(pi_C(p), s)),
                { r: ActionTransition |
                        r in R(pi_1(p), s)
                     && is_Checkmark(pi_t(r))
                     && pi_sigma'(r) == s, {});

% (p,s) —m— -> (p',s') && [[b]](s) == true
% ————————————————————————————————————
% (b -> p <> q,s) —m— -> (p',s')
R_Cond2_2(p,s)= if(is_Cond2(p)
    && Cast2InternalBool(Sem_Dex(pi_C(p), s)),
               { r: ActionTransition |
                       r in R(pi_1(p), s)
                    && !is_Checkmark(pi_t(r))}, {});

% (q,s) —m—> Checkmark && [[b]](s) == false
% ————————————————————————————————————
% (b -> p <> q,s) —m—> Checkmark
R_Cond2_3(p,s)= if(is_Cond2(p)
    && !Cast2InternalBool(Sem_Dex(pi_C(p), s)),
                { r: ActionTransition |
                       r in R(pi_2(p), s)
                    && is_Checkmark(pi_t(r))
                    && pi_sigma'(r) == s}, {});

% (q,s) —m— -> (q',s') && [[b]](s) == false
% ————————————————————————————————————
% (b -> p <> q,s) —m— -> (q',s')
R_Cond2_4(p,s)= if(is_Cond2(p)
    && !Cast2InternalBool(Sem_Dex(pi_C(p), s)),
                { r: ActionTransition |
                       r in R(pi_2(p), s)
                    && !is_Checkmark(pi_t(r))}, {});

% (p,s[d: =e]) —m—> Checkmark
% ———————————————————————————— e in M_D
% (sum d: D . p, s) —m—> Checkmark
R_Sum_1(p,s) = if(is_Sum(p), { r: ActionTransition |
                    pi_sigma'(r) == s
                 && is_Checkmark(pi_t(r))
                 &&(exists v: Value.
                       RestrictDomain(pi_v(p), v)
                    &&(at(pi_ac(r), pi_t(r), Z) in R(pi_1(p), Z)
                      whr Z = InsertArgument(argument(pi_v(p), v), s) end)}, {});

% (p[d : = d' ],s[d': =e]) —m— -> (p',s')
% ———————————————————————————— e in M_D
% (sum d: D . p, s) —m— -> (p',s')
R_Sum_2(p,s) = if(is_Sum(p), { r: ActionTransition |
                    !is_Checkmark(pi_t(r))
                 &&(exists v: Value.
                    RestrictDomain(pi_v(p), v)
                 && r in R(VariableSubstitutionInProcessTerm(
                           (lambda v: Variable.(v))[ pi_v(p) -> VAR ], pi_1(p)),
                              InsertArgument(argument(VAR, v), s)))}
                    whr VAR= GenFreshVar(pi_v(p), GetHighestId(s) + 1) end, {});

% (p,s) —m—> Checkmark
% ————————————————————
% (p || q, s) —m— -> (q, s)
R_Par_1(p,s) = if(is_Par(p), { r: ActionTransition |
                    at(pi_ac(r), Checkmark, s) in R(pi_1(p), s)
                 && pi_t(r) == pi_2(p)
```

```
                    && pi_sigma'(r) == s}, {});

    % (p,s) --m-- -> (p',s')
    % ----------------------------------
    % (p || q, s) --m-- -> (p'|| q, s')
    R_Par_2(p,s) = if(is_Par(p), { r: ActionTransition |
                        is_Par(pi_t(r))
                    && at(pi_ac(r), pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)
                    && !is_Checkmark(pi_1(pi_t(r)))
                    && pi_2(pi_t(r)) == pi_2(p)}, {});

    % (q,s) --m--> Checkmark
    % ----------------------------
    % (p || q, s) --m-- -> (p, s)
    R_Par_3(p,s) = if(is_Par(p), { r: ActionTransition |
                        at(pi_ac(r), Checkmark, s) in R(pi_2(p), s)
                    && pi_t(r) == pi_1(p)
                    && pi_sigma'(r) == s}, {});

    % (p,s) --m-- -> (q',s')
    % ----------------------------------
    % (p || q, s) --m-- -> (p || q', s')
    R_Par_4(p,s) = if(is_Par(p), { r: ActionTransition |
                        is_Par(pi_t(r))
                    && at(pi_ac(r), pi_2(pi_t(r)), pi_sigma'(r)) in R(pi_2(p), s)
                    && !is_Checkmark(pi_2(pi_t(r)))
                    && pi_1(pi_t(r)) == pi_1(p)}, {});

    % (q,s) --m--> Checkmark,(q,s) --n--> Checkmark
    % ------------------------------------------
    % (p || q, s) --m|n--> Checkmark
    R_Par_5(p,s) = if(is_Par(p), { r: ActionTransition |
                        is_Checkmark(pi_t(r))
                    && pi_sigma'(r) == s
                    && exists t_1, t_2: List(ActionSemantic).
                        at(t_1, Checkmark, s) in R(pi_1(p), s)
                    && at(t_2, Checkmark, s) in R(pi_2(p), s)
                    && MergeOrderedActionLists(t_1, t_2) == pi_ac(r)}, {});

    % (q,s) --m-- -> (p', s'),(q,s) --n--> Checkmark
    % ------------------------------------------
    % (p || q, s) --m|n-- -> (p',s')
    R_Par_6(p,s) = if(is_Par(p), { r: ActionTransition |
                        exists r_1, r_2: ActionTransition.
                           r_1 in R(pi_1(p), s)
                        && r_2 in R(pi_2(p), s)
                        && is_Checkmark(pi_t(r_1))
                        && !is_Checkmark(pi_t(r_2))
                        && MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2)) == pi_ac(r)
                        && pi_t(r) == pi_t(r_2)
                        && pi_sigma'(r) == pi_sigma'(r_2)}, {});

    % (q,s) --m--> Checkmark,(q,s) --n-- -> (q',s')
    % ------------------------------------------
    % (p || q, s) --m|n-- -> (q',s')
    R_Par_7(p,s) = if(is_Par(p), { r: ActionTransition |
                        exists r_1, r_2: ActionTransition.
                           r_1 in R(pi_1(p), s)
                        && r_2 in R(pi_2(p), s)
                        && !is_Checkmark(pi_t(r_1))
                        && is_Checkmark(pi_t(r_2))
                        && MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2)) == pi_ac(r)
                        && pi_t(r) == pi_t(r_1)
                        && pi_sigma'(r) == pi_sigma'(r_1)}, {});
```

```
315     % (q,s) —m— -> (p',s'),(q,s) —n— -> (q',s'')
316     % ————————————————————————————————————————
317     % (p || q, s) —m|n— -> (p'||q', s' ++ s'')
318     R_Par_8(p,s) = if(is_Par(p), { r: ActionTransition |
319                        is_Par(pi_t(r))
320                    && exists r_1, r_2: ActionTransition.(
321                          r_1 in R(pi_1(p), s)
322                      && r_2 in R(pi_2(p), s)
323                      && !is_Checkmark(pi_t(r_1))
324                      && !is_Checkmark(pi_t(r_2))
325                      && pi_ac(r) == MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2))
326                      && pi_1(pi_t(r)) == pi_t(r_1)
327                      && pi_2(pi_t(r)) ==
328                            VariableSubstitutionInProcessTerm(SUBST, pi_t(r_2))
329                      && pi_sigma'(r)  ==
330                            MergeOrderedValuations(pi_sigma'(r_1),
331                               VariableSubstitutionInValuation(SUBST,
332                                 ValuationMinusValuationOrdered(pi_sigma'(r_2),s)))
333                    whr SUBST =
334                      CreateVariableSubstitution(
335                        DUP, GenFreshVars(
336                        max(GetHighestId(pi_sigma'(r_1)),
337                            GetHighestId(pi_sigma'(r_2)))+ 1, DUP))
338                    whr DUP = DuplicateVariablesInValuationOrdered(
339                          ValuationMinusValuationOrdered(pi_sigma'(r_2),s),
340                          ValuationMinusValuationOrdered(pi_sigma'(r_1),s)) end
341                    end)}, {});
342
343     % (p,s) —m—> Checkmark
344     % ————————————————————
345     % (p ||_ q, s) —m— -> (q, s)
346     R_Lmerge_1(p,s) = if(is_Lmerge(p), { r: ActionTransition |
347                          at(pi_ac(r), Checkmark, s) in R(pi_1(p),s)
348                        && pi_t(r) == pi_2(p)
349                        && pi_sigma'(r) == s}, {});
350
351     % (p,s) —m— -> (p',s')
352     % ————————————————————————————————
353     % (p ||_ q, s) —m— -> (p' || q, s')
354     R_Lmerge_2(p,s) = if(is_Lmerge(p), { r: ActionTransition |
355                          is_Par(pi_t(r))
356                        && at(pi_ac(r), pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p),s)
357                        && pi_2(pi_t(r)) == pi_2(p)
358                        && pi_1(pi_t(r)) != Checkmark}, {});
359
360     % (p,s) —m—> Checkmark,(q,s) —n—> Checkmark
361     % ————————————————————————————————————————
362     % (p|q, s) —m|n—> Checkmark
363     R_Sync_1(p,s) = if(is_Sync(p), { r: ActionTransition |
364                        pi_sigma'(r) == s
365                    && is_Checkmark(pi_t(r))
366                    && exists r_1, r_2: ActionTransition.
367                        r_1 in R(pi_1(p), s)
368                    && r_2 in R(pi_2(p), s)
369                    && is_Checkmark(pi_t(r_1))
370                    && is_Checkmark(pi_t(r_2))
371                    && MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2)) == pi_ac(r)}, {});
372
373     % (p,s) —m— -> (p',s'),(q,s) —n—> Checkmark
374     % ————————————————————————————————————————
375     % (p|q, s) —m|n— -> (p',s')
376     R_Sync_2(p,s) = if(is_Sync(p), { r: ActionTransition |
377                        exists r_1, r_2: ActionTransition.
378                          r_1 in R(pi_1(p), s)
379                        && r_2 in R(pi_2(p), s)
```

```
                        && !is_Checkmark(pi_t(r_1))
                        && is_Checkmark(pi_t(r_2))
                        && MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2)) == pi_ac(r)
                        && pi_t(r) == pi_t(r_1)
                        && pi_sigma'(r_1) == pi_sigma'(r)}, {});

    % (p,s) --m--> Checkmark,(q,s) --n- -> (q',s')
    % ---------------------------------------------
    % (p|q, s) --m|n- -> (q',s')
    R_Sync_3(p,s) = if(is_Sync(p), { r: ActionTransition |
                        exists r_1, r_2: ActionTransition.
                           r_1 in R(pi_1(p), s)
                        && r_2 in R(pi_2(p), s)
                        && is_Checkmark(pi_t(r_1))
                        && !is_Checkmark(pi_t(r_2))
                        && MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2)) == pi_ac(r)
                        && pi_t(r) == pi_t(r_2)
                        && pi_sigma'(r) == pi_sigma'(r_1)}, {});

    % (p,s) --m- -> (p,s'),(q,s) --n- -> (q',s'')
    % ---------------------------------------------
    % (p|q, s) --m|n- -> (p'||q',s'++s'')
    R_Sync_4(p,s) = if(is_Sync(p), { r: ActionTransition |
                          is_Par(pi_t(r))
                       && exists r_1, r_2: ActionTransition.(
                           r_1 in R(pi_1(p), s)
                        && r_2 in R(pi_2(p), s)
                        && !is_Checkmark(pi_t(r_1))
                        && !is_Checkmark(pi_t(r_2))
                        && pi_ac(r) == MergeOrderedActionLists(pi_ac(r_1), pi_ac(r_2))
                        && pi_1(pi_t(r)) == pi_t(r_1)
                        && pi_2(pi_t(r)) ==
                                VariableSubstitutionInProcessTerm(SUBST, pi_t(r_2))
                        && pi_sigma'(r)  ==
                                MergeOrderedValuations(pi_sigma'(r_1),
                                  VariableSubstitutionInValuation(SUBST,
                                    ValuationMinusValuationOrdered(pi_sigma'(r_2),s)))
                       whr SUBST =
                         CreateVariableSubstitution(
                           DUP, GenFreshVars(
                           max(GetHighestId(pi_sigma'(r_1)),
                               GetHighestId(pi_sigma'(r_2)))+ 1, DUP))
                       whr DUP = DuplicateVariablesInValuationOrdered(
                            ValuationMinusValuationOrdered(pi_sigma'(r_2),s),
                            ValuationMinusValuationOrdered(pi_sigma'(r_1),s)) end
                       end)}, {});

    % (p,s) --m--> Checkmark,(m _in_ V +{tau})
    % ---------------------------------------------
    % (allow(A,p),s) --m--> Checkmark
    R_Allow_1(p,s) = if(is_Allow(p), { r: ActionTransition |
                          pi_t(r)  == Checkmark
                        && r in R(pi_1(p), s)
                        && ActionLabels(pi_ac(r)) in (pi_V(p) + {{}})
                        && pi_sigma'(r) == s}, {});

    % (p,s) --m- -> (p',s'),(m _in_ V +{tau})
    % ---------------------------------------------
    % (allow(A,p),s) --m- -> (allow(A,p'),s')
    R_Allow_2(p,s) = if(is_Allow(p), { r: ActionTransition |
                          is_Allow(pi_t(r))
                        && pi_1(pi_t(r)) != Checkmark
                        && pi_V(pi_t(r)) == pi_V(p)
                        && at(pi_ac(r),pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)
                        && ActionLabels(pi_ac(r)) in (pi_V(p) + {{}})}, {});
```

```
445
446     % (p,s) --m--> Checkmark,(m{}    B == {})
447     % ------------------------------------------
448     % (block(B,p),s) --m--> Checkmark
449     is_Block_1(p,s) = if(is_Block(p), { r: ActionTransition |
450                           pi_t(r)  == Checkmark
451                        && r in R(pi_1(p), s)
452                        && Bag2Set(ActionLabels(pi_ac(r)))    pi_B(p) == {}
453                        && pi_sigma'(r) == s}, {});
454
455     % (p,s) --m--> Checkmark,(m{}    B == {})
456     % ------------------------------------------
457     % (block(B,p),s) --m- -> (allow(B,p'),s')
458     is_Block_2(p,s) = if(is_Block(p), { r: ActionTransition |
459                           is_Block(pi_t(r))
460                        && pi_1(pi_t(r))  != Checkmark
461                        && pi_B(pi_t(r)) == pi_B(p)
462                        && at(pi_ac(r),pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)
463                        && Bag2Set(ActionLabels(pi_ac(r)))    pi_B(p) == {}}, {});
464
465     % (p,s) --m--> Checkmark
466     % ------------------------------------------
467     % (rename(R,p),s) --R(m)--> Checkmark
468     R_Rename_1(p,s) = if(is_Rename(p), { r: ActionTransition |
469                           is_Checkmark(pi_t(r))
470                        && exists ac': List(ActionSemantic).
471                              pi_ac(r) == ActRename(pi_Ren(p), ac')
472                           && at(ac', pi_t(r), s) in R(pi_1(p), s)
473                           && pi_sigma'(r) == s}, {});
474
475     % (p,s) --m- -> (p',s')
476     % ------------------------------------------
477     % (rename(R,p),s) --Ren(R,m)- -> (rename(R,p'),s')
478     R_Rename_2(p,s) = if(is_Rename(p), { r: ActionTransition |
479                           pi_Ren(pi_t(r)) == pi_Ren(p)
480                        && is_Rename(pi_t(r))
481                        && pi_1(pi_t(r)) != Checkmark
482                        && exists ac': List(ActionSemantic).
483                              pi_ac(r) == ActRename(pi_Ren(p), ac')
484                           && at(ac',pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)}, {});
485
486     % (p,s) --m--> Checkmark
487     % ------------------------------------------
488     % (hide(I,p),s) --h(I,m)--> Checkmark
489     R_Hide_1(p,s) = if(is_Hide(p), { r: ActionTransition |
490                           is_Checkmark(pi_t(r))
491                        && exists ac': List(ActionSemantic).
492                              pi_ac(r) == ActHide(pi_I(p), ac')
493                           && at(ac', pi_t(r), s) in R(pi_1(p), s)
494                           && pi_sigma'(r) == s}, {});
495
496     % (p,s) --m- -> (p',s')
497     % ------------------------------------------
498     % (hide(I,p),s) --h(I,m)- -> (hide(I,p'),s')
499     R_Hide_2(p,s) = if(is_Hide(p), { r: ActionTransition |
500                           pi_I(pi_t(r)) == pi_I(p)
501                        && is_Hide(pi_t(r))
502                        && pi_1(pi_t(r)) != Checkmark
503                        && exists ac': List(ActionSemantic).
504                              pi_ac(r) == ActHide(pi_I(p), ac')
505                           && at(ac',pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)}, {});
506
507    % (p,s) --m--> Checkmark
508    % ------------------------------------------
509    % (prehide(U,p),s) --ph(U,m)--> Checkmark
```

```
510     R_Prehide_1(p,s) = if(is_Prehide(p), { r: ActionTransition |
511                            is_Checkmark(pi_t(r))
512                         && exists ac': List(ActionSemantic).
513                               pi_ac(r) == ActPrehide(pi_U(p), ac')
514                            && at(ac', pi_t(r), s) in R(pi_1(p), s)
515                            && pi_sigma'(r) == s}, {});
516
517     % (p,s) --m-- -> (p',s')
518     % ------------------------------------------------------
519     % (prehide(U,p),s) --ph(U,m)-- -> (prehide(U,p'),s')
520     R_Prehide_2(p,s) = if(is_Prehide(p), { r: ActionTransition |
521                            pi_U(pi_t(r)) == pi_U(p)
522                         && is_Prehide(pi_t(r))
523                         && pi_1(pi_t(r)) != Checkmark
524                         && exists ac': List(ActionSemantic).
525                              pi_ac(r) == ActPrehide(pi_U(p), ac')
526                            && at(ac',pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)}, {});
527
528     % (p,s) --m--> Checkmark
529     % ------------------------------------
530     % (comm(C,p),s) --cm(C,m)--> Checkmark
531     R_Comm_1(p,s) = if(is_Comm(p), { r: ActionTransition |
532                         is_Checkmark(pi_t(r))
533                      && exists ac': List(ActionSemantic).
534                            at(ac', pi_t(r), s) in R(pi_1(p), s)
535                         && pi_sigma'(r) == s
536                         && pi_ac(r) ==  ActComm(pi_CL(p), ac')}, {});
537
538     % (p,s) --m-- -> (p',s')
539     % ------------------------------------------------
540     % (comm(C,p),s) --cm(C,m)-- -> (comm(C,p'),s')
541     R_Comm_2(p,s) = if(is_Comm(p), { r: ActionTransition |
542                            pi_CL(pi_t(r)) == pi_CL(p)
543                         && is_Comm(pi_t(r))
544                         && pi_1(pi_t(r)) != Checkmark
545                         && exists ac': List(ActionSemantic).
546                               pi_ac(r) == ActComm(pi_CL(p), ac')
547                            && at(ac',pi_1(pi_t(r)), pi_sigma'(r)) in R(pi_1(p), s)}, {});
548
549     % (q,s(d: =[[e]](s)) --m--> Checkmark
550     % ------------------------------------------
551     % (P(e),s) --m--> Checkmark
552     R_Def_1(p, s) = if(is_Def(p), { r: ActionTransition |
553                            at(pi_ac(r), pi_t(r), Z) in R(Def(pi_P(p)), Z)
554                         && pi_sigma'(r) == s
555                         && is_Checkmark(pi_t(r))}, {})
556                         whr Z =
557                              MergeOrderedValuations(
558                              ComputePPunderValuation(ppl(p), s),
559                                RemoveArgumentWithDuplicateVariable(ppl(p), s))
560                         end;
561
562
563     % (q[d: =d'],s(d': =[[e]](s)) --m-- -> (q',s')
564     % ------------------------------------------------------
565     % (P(e),s) --m-- -> (q',s')
566     R_Def_2(p, s) = if(is_Def(p), { r: ActionTransition |
567                            r in R(SUBST, MergeOrderedValuations(REN,s))
568                         && !is_Checkmark(pi_t(r))}, {})
569                         whr REN = VariableSubstitutionInValuation(
570                                CreateVariableSubstitution(
571                                  GetVarLabelsFromPP(ppl(p)),
572                                  GenFreshVars(GetHighestId(s) + 1,
573                                    GetVarLabelsFromPP(ppl(p)))),
574                                ComputePPunderValuation(ppl(p),
```

```
                                s)),
                            SUBST = VariableSubstitutionInProcessTerm(
                              CreateVariableSubstitution(
                                GetVarLabelsFromPP(ppl(p)),
                                GenFreshVars(GetHighestId(s) + 1,
                                  GetVarLabelsFromPP(ppl(p)))),
                              Def(pi_P(p)))
                        end;

% Semantic interpretation function
map Sem_ActList: List(ActionSyntax)#Valuation -> List(ActionSemantic);
    Sem_Act: ActionSyntax# Valuation -> ActionSemantic;
    Sem_DexList: List(DataExpression)#Valuation -> List(Value);
var a: ActionSyntax;
    as: List(ActionSyntax);
    sigma: Valuation;
    d: Variable;
    des: List(DataExpression);
    de: DataExpression;
    expr_1: DataExpression;
    expr_2: DataExpression;
    l: ActionLabel;
eqn Sem_ActList([], sigma)       = [];
    Sem_ActList(a |> as, sigma)  =
      if(a == ActionTau,
        Sem_ActList(as, sigma),
        InsertAction(Sem_Act(a, sigma), Sem_ActList(as, sigma)));
    Sem_Act(Act(l,des), sigma) = ActSem(l, Sem_DexList(des, sigma));
    Sem_DexList([], sigma) = [];
    Sem_DexList(de |> des, sigma) =
        Sem_Dex(de, sigma) |> Sem_DexList(des, sigma);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Auxiliary functions for the deduction rules %%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Label identity function
map ID: ActionLabel -> ActionLabel;
var x: ActionLabel;
eqn ID(x)=x;

% Action rename function
map ActRename: (ActionLabel -> ActionLabel)#List(ActionSemantic)
                -> List(ActionSemantic);
var f: ActionLabel -> ActionLabel;
    a: ActionSemantic;
    as: List(ActionSemantic);
eqn ActRename(f, []) = [];
    ActRename(f, a |> as) =
      InsertAction(ActSem(f(ActionLabel(a)), args(a)), ActRename(f, as));

% Action hide function
map ActHide: Set(ActionLabel)#List(ActionSemantic) -> List(ActionSemantic);
var I: Set(ActionLabel);
    as: List(ActionSemantic);
    a: ActionSemantic;
eqn ActHide(I, [])       = [];
    ActHide(I, a |> as) =
                  if(ActionLabel(a) in I, ActHide(I, as), a |> ActHide(I, as));

% Action prehide function
map ActPrehide: Set(ActionLabel)#List(ActionSemantic) -> List(ActionSemantic);
var U: Set(ActionLabel);
    as: List(ActionSemantic);
    a: ActionSemantic;
```

```
640  eqn ActPrehide(U, [])        = [];
641      ActPrehide(U, a |> as) =
642        if(ActionLabel(a) in U, InsertAction(ActSem(int, []), ActPrehide(U, as))
643                              , a |> ActPrehide(U, as));
644
645  % Action communication function
646  map ActComm: List(Communication)#List(ActionSemantic) -> List(ActionSemantic);
647  var as: List(ActionSemantic);
648      C: Communication;
649      CL: List(Communication);
650  eqn ActComm([]     , as) = as;
651      ActComm(C |> CL, as) =
652        ActCommAux(C, CL, as,
653          ArgumentsToActionLabelMap(as, lambda x: List(Value). {}),
654          ActionLabelsToBag(CmI(C)), [], []);
655
656
657  % Auxiliary function required by the action communication function
658  % that maps action data parameters to a bag of action labels
659  map ArgumentsToActionLabelMap: List(ActionSemantic)#
660      (List(Value) -> Bag(ActionLabel)) ->(List(Value) -> Bag(ActionLabel));
661  var ActionParametersToActionLabels: List(Value) -> Bag(ActionLabel);
662      as: List(ActionSemantic);
663      a: ActionSemantic;
664  eqn ArgumentsToActionLabelMap([], ActionParametersToActionLabels) =
665        ActionParametersToActionLabels;
666      ArgumentsToActionLabelMap(a |> as, ActionParametersToActionLabels) =
667        ArgumentsToActionLabelMap(as,
668          ActionParametersToActionLabels[
669            args(a) -> ActionParametersToActionLabels(args(a)) +
670                       {ActionLabel(a): 1}]);
671
672  % Auxiliary function required by the action communication function
673  % that transforms a list of action labels to a bag of action labels.
674  map ActionLabelsToBag: List(ActionLabel) -> Bag(ActionLabel);
675  var ActLab: ActionLabel;
676      CommActLabels: List(ActionLabel);
677  eqn ActionLabelsToBag([]) = {};
678      ActionLabelsToBag(ActLab |> CommActLabels) =
679        {ActLab: 1} + ActionLabelsToBag(CommActLabels);
680
681  % Auxiliary function required by the action communication function
682  % that computes the Synchronizing actions in the remaining multi-action.
683  map ActCommAux: Communication#List(Communication)#List(ActionSemantic)#
684      (List(Value) -> Bag(ActionLabel))#Bag(ActionLabel)#List(ActionSemantic)#
685        List(ActionSemantic) -> List(ActionSemantic);
686  var ActionParametersToActionLabels: List(Value) -> Bag(ActionLabel);
687      CommActBag: Bag(ActionLabel);
688      ResultActions, RemainingActions: List(ActionSemantic);
689      C: Communication;
690      CL: List(Communication);
691      as: List(ActionSemantic);
692      a: ActionSemantic;
693  eqn ActCommAux(C, CL, [], ActionParametersToActionLabels,
694        CommActBag, ResultActions, RemainingActions) =
695          MergeOrderedActionLists(ResultActions, ActComm(CL, RemainingActions));
696      ActCommAux(C, CL, a |> as, ActionParametersToActionLabels,
697        CommActBag, ResultActions, RemainingActions) =
698          if(CommActBag <= ActionParametersToActionLabels(args(a)),
699            %Condition holds
700            ActCommAux(C, CL,
701              EliminateMatchingActions(CmI(C), a |> as, args(a)),
702              ArgumentsToActionLabelMap(
703                EliminateMatchingActions(CmI(C), a |> as, args(a)),
704                lambda x: List(Value). {}),
```

```
705              CommActBag, InsertAction(ActSem(CmR(C), args(a)), ResultActions),
706              RemainingActions),
707          %Condition does not hold
708           ActCommAux(C, CL, as, ActionParametersToActionLabels, CommActBag,
709              ResultActions, a |> RemainingActions));
710
711 % Auxiliary function required by the action communication function
712 % to remove an occurrence of a Synchonizing action in the remaining multi-action.
713 map EliminateMatchingActions: List(ActionLabel)#List(ActionSemantic)#List(Value)
714       -> List(ActionSemantic);
715 var ActLab: ActionLabel;
716     CommActLabels: List(ActionLabel);
717     as: List(ActionSemantic);
718     args: List(Value);
719 eqn EliminateMatchingActions([], as, args) = as;
720     EliminateMatchingActions(ActLab |> CommActLabels, as, args) =
721       EliminateMatchingActions(CommActLabels,
722         RemoveAction(ActSem(ActLab,args), as), args);
723
724 % Auxiliary function that is required by the action communication function
725 % to remove an action in the remainder of a list of actions, i.e. a multi-action.
726 map RemoveAction: ActionSemantic#List(ActionSemantic) -> List(ActionSemantic);
727 var a, b: ActionSemantic;
728     as: List(ActionSemantic);
729 eqn RemoveAction(a, []) = [];
730     RemoveAction(a, b |> as) = if(a == b, as, b |> RemoveAction(a, as));
731
732 % Determine the last generated fresh variable in a data valuation.
733 map GetHighestId: Valuation -> Nat;
734     GetVarId : Argument -> Nat;
735 var las: Valuation;
736     a: Argument;
737 eqn GetHighestId([]) = 0;
738     GetHighestId(a |> las) = max(GetVarId(a), GetHighestId(las));
739     GetVarId(a) =
740       if(is_d'(variablelabel(variable(a))),
741         id(variablelabel(variable(a))), 0);
742
743 % Conversion of a list of semantics action to a bag of action labels.
744 map ActionLabels: List(ActionSemantic) -> Bag(ActionLabel);
745 var as: List(ActionSemantic);
746     a: ActionSemantic;
747 eqn ActionLabels([]) = {};
748     ActionLabels(a |> as) = {ActionLabel(a): 1} + ActionLabels(as);
749
750 % Function that retrieves the variables from the process parameters.
751 map GetVarLabelsFromPP: List(PP) -> List(Variable);
752 var ppl: List(PP);
753     pp : PP;
754 eqn GetVarLabelsFromPP([]) = [];
755     GetVarLabelsFromPP(pp |> ppl) = variable(pp) |> GetVarLabelsFromPP(ppl);
756
757 % Function that generates new variables for a vector of variables.
758 % Identifier starts at value of 'n'
759 map GenFreshVars: Nat#List(Variable) -> List(Variable);
760 var vs: List(Variable);
761     v : Variable;
762     n: Nat;
763 eqn GenFreshVars(n,[])        =  [];
764     GenFreshVars(n, v |> vs) =  GenFreshVar(v,n) |> GenFreshVars(n+1, vs);
765
766 % Function that transforms a list of process parameters into a valuation.
767 map ComputePPunderValuation: PP#Valuation -> Argument;
768     ComputePPunderValuation: List(PP)#Valuation -> Valuation;
769 var p: PP;
```

```
770      pl: List(PP);
771      s: Valuation;
772  eqn ComputePPunderValuation(p,s) =
773        argument(variable(p), Sem_Dex(dataexpression(p), s));
774      ComputePPunderValuation([], s) = [];
775      ComputePPunderValuation(p |> pl, s) =
776        InsertArgument(ComputePPunderValuation(p,s), ComputePPunderValuation(pl,s));
777
778  % Function that creates a variable substitution.
779  map CreateVariableSubstitution:
780        List(Variable)#List(Variable) ->(Variable -> Variable);
781      CreateVariableSubstitution:
782        List(Variable)#List(Variable)#(Variable -> Variable) ->
783        (Variable -> Variable);
784  var OldVar: Variable;
785      NewVar: Variable;
786      OldVars: List(Variable);
787      NewVars: List(Variable);
788      VarRename: Variable -> Variable;
789  eqn CreateVariableSubstitution(OldVars, NewVars) =
790        CreateVariableSubstitution(OldVars, NewVars, lambda v: Variable.(v));
791      CreateVariableSubstitution([], [], VarRename)  = VarRename;
792      CreateVariableSubstitution(OldVar |> OldVars, NewVar |> NewVars, VarRename)=
793        CreateVariableSubstitution(OldVars, NewVars, VarRename[OldVar -> NewVar]);
794
795  % Function that substitutes variables in a valuation.
796  map VariableSubstitutionInValuation: (Variable -> Variable)#Valuation-> Valuation;
797  var VarRename: Variable -> Variable;
798      as: Valuation;
799      a : Argument;
800  eqn VariableSubstitutionInValuation(VarRename, []) = [];
801      VariableSubstitutionInValuation(VarRename, a |> as) =
802        InsertArgument(argument(VarRename(variable(a)), valvalue(a)),
803        VariableSubstitutionInValuation(VarRename,as));
804
805  % Function that substitutes variables in a process term.
806  map VariableSubstitutionInProcesParameters:
807        (Variable -> Variable)#List(PP) -> List(PP);
808      VariableSubstitutionInProcessTerm:
809        (Variable -> Variable)#ProcessTerm -> ProcessTerm;
810      VariableSubstitutionInActionList:
811        (Variable -> Variable)#List(ActionSyntax) -> List(ActionSyntax);
812      VariableSubstitutionInAction:
813        (Variable -> Variable)#ActionSyntax -> ActionSyntax;
814      VariableSubstitutionInVariableList:
815        (Variable -> Variable)#List(Variable) -> List(Variable);
816      VariableSubstitutionInDataExpressionList:
817        (Variable -> Variable)#List(DataExpression) -> List(DataExpression);
818  var VarRename: Variable -> Variable;
819      v: Variable;
820      pt1, pt2: ProcessTerm;
821      al: List(ActionSyntax);
822      a: ActionSyntax;
823      vl: List(Variable);
824      dl: List(DataExpression);
825      d: DataExpression;
826      ppl: List(PP);
827      pp: PP;
828      V: Set(Bag(ActionLabel));
829      B: Set(ActionLabel);
830      Ren: ActionLabel -> ActionLabel;
831      I,U: Set(ActionLabel);
832      CmI: List(ActionLabel);
833      CmR: ActionLabel;
834      P: ProcessLabel;
```

```
835      CL: List(Communication);
836  eqn VariableSubstitutionInVariableList(VarRename, []) = [];
837      VariableSubstitutionInVariableList(VarRename, v |> vl) =
838        VarRename(v) |> VariableSubstitutionInVariableList(VarRename, vl);
839      VariableSubstitutionInDataExpressionList(VarRename, []) = [];
840      VariableSubstitutionInDataExpressionList(VarRename, d |> dl) =
841        VariableSubstitutionInDataExpression(VarRename, d) |>
842          VariableSubstitutionInDataExpressionList(VarRename, dl);
843      VariableSubstitutionInAction(VarRename, a) =
844        Act(ActionLabel(a),
845          VariableSubstitutionInDataExpressionList(VarRename, args(a)));
846      VariableSubstitutionInActionList(VarRename, []) = [];
847      VariableSubstitutionInActionList(VarRename, a |> al) =
848        VariableSubstitutionInAction(VarRename, a) |>
849          VariableSubstitutionInActionList(VarRename, al);
850      VariableSubstitutionInProcessTerm(VarRename, Deadlock) = Deadlock;
851      VariableSubstitutionInProcessTerm(VarRename, Checkmark) = Checkmark;
852      VariableSubstitutionInProcessTerm(VarRename, Alpha(al)) =
853        Alpha(VariableSubstitutionInActionList(VarRename, al));
854      VariableSubstitutionInProcessTerm(VarRename, Seq(pt1, pt2)) =
855        Seq(VariableSubstitutionInProcessTerm(VarRename, pt1),
856          VariableSubstitutionInProcessTerm(VarRename, pt2));
857      VariableSubstitutionInProcessTerm(VarRename, Alt(pt1, pt2)) =
858        Alt(VariableSubstitutionInProcessTerm(VarRename, pt1),
859          VariableSubstitutionInProcessTerm(VarRename, pt2));
860      VariableSubstitutionInProcessTerm(VarRename, Cond1(d, pt1)) =
861        Cond1(VariableSubstitutionInDataExpression(VarRename, d),
862          VariableSubstitutionInProcessTerm(VarRename, pt1));
863      VariableSubstitutionInProcessTerm(VarRename, Cond2(d, pt1, pt2)) =
864        Cond2(VariableSubstitutionInDataExpression(VarRename, d),
865          VariableSubstitutionInProcessTerm(VarRename, pt1),
866          VariableSubstitutionInProcessTerm(VarRename, pt2));
867      VariableSubstitutionInProcessTerm(VarRename, Sum(v, pt1)) =
868        Sum(VarRename(v), VariableSubstitutionInProcessTerm(VarRename, pt1));
869      VariableSubstitutionInProcessTerm(VarRename, Par(pt1, pt2)) =
870        Par(VariableSubstitutionInProcessTerm(VarRename, pt1),
871          VariableSubstitutionInProcessTerm(VarRename, pt2));
872      VariableSubstitutionInProcessTerm(VarRename, Sync(pt1, pt2)) =
873        Sync(VariableSubstitutionInProcessTerm(VarRename, pt1),
874          VariableSubstitutionInProcessTerm(VarRename, pt2));
875      VariableSubstitutionInProcessTerm(VarRename, Allow(V, pt1)) =
876        Allow(V, VariableSubstitutionInProcessTerm(VarRename, pt1));
877      VariableSubstitutionInProcessTerm(VarRename, Block(B, pt1)) =
878        Block(B, VariableSubstitutionInProcessTerm(VarRename, pt1));
879      VariableSubstitutionInProcessTerm(VarRename, Hide(I, pt1)) =
880        Hide(I, VariableSubstitutionInProcessTerm(VarRename, pt1));
881      VariableSubstitutionInProcessTerm(VarRename, Prehide(U, pt1)) =
882        Prehide(U, VariableSubstitutionInProcessTerm(VarRename, pt1));
883      VariableSubstitutionInProcessTerm(VarRename, Comm(CL, pt1))=
884        Comm(CL, VariableSubstitutionInProcessTerm(VarRename, pt1));
885      VariableSubstitutionInProcessTerm(VarRename, Def(P, ppl)) =
886        Def(P, VariableSubstitutionInProcesParameters(VarRename, ppl));
887      VariableSubstitutionInProcessTerm(VarRename, Rename(Ren, pt1)) =
888        Rename(Ren, VariableSubstitutionInProcessTerm(VarRename, pt1));
889      VariableSubstitutionInProcesParameters(VarRename, []) = [];
890      VariableSubstitutionInProcesParameters(VarRename, pp |> ppl) =
891        pp(variable(pp),
892          VariableSubstitutionInDataExpression(VarRename, dataexpression(pp))) |>
893            VariableSubstitutionInProcesParameters(VarRename, ppl);
894
895  % Function that retrieves duplicate variables from a valuation.
896  map DuplicateVariablesInValuationOrdered: Valuation#Valuation -> List(Variable);
897  var x,y: Argument;
898      xs: Valuation;
899      ys: Valuation;
```

```
900  eqn DuplicateVariablesInValuationOrdered([], ys) = [];
901      DuplicateVariablesInValuationOrdered(xs, []) = [];
902      DuplicateVariablesInValuationOrdered(x |> xs, y |> ys) =
903        if(variable(x) == variable(y),
904          variable(x) |> DuplicateVariablesInValuationOrdered(xs, ys),
905          if(variable(x) < variable(y),
906              DuplicateVariablesInValuationOrdered(xs, y |> ys),
907              DuplicateVariablesInValuationOrdered(x |> xs, ys)));
908
909  % Function that inserts an argument into a valuation.
910  map InsertArgument : Argument# Valuation -> Valuation;
911  var x,y: Argument;
912      ys: Valuation;
913  eqn InsertArgument(x, []) = [x];
914      InsertArgument(x, y|> ys) =
915        if(x <= y, x|> y|> ys, y|> InsertArgument(x, ys));
916
917  % Function that inserts a semantic action into a list of semantic actions.
918  map InsertAction:
919        ActionSemantic# List(ActionSemantic) -> List(ActionSemantic);
920  var x,y: ActionSemantic;
921      ys: List(ActionSemantic);
922  eqn InsertAction(x, []) = [x];
923      InsertAction(x, y|> ys)= if(x <=y, x |> y |> ys, y |> InsertAction(x, ys));
924
925
926  % Merge two ordered lists of semantic actions.
927  map MergeOrderedActionLists: List(ActionSemantic)#List(ActionSemantic)
928        -> List(ActionSemantic);
929  var x,y: ActionSemantic;
930      xs,ys: List(ActionSemantic);
931  eqn MergeOrderedActionLists([],[])       = [];
932      MergeOrderedActionLists([],xs)       = xs;
933      MergeOrderedActionLists(xs,[])       = xs;
934      MergeOrderedActionLists(x |> xs, y |> ys) =
935        if(x <= y, x |> MergeOrderedActionLists(xs, y |> ys),
936          y |> MergeOrderedActionLists(x |> xs, ys));
937
938  % Merge two ordered valuations.
939  map MergeOrderedValuations: Valuation#Valuation -> Valuation;
940  var x,y: Argument;
941      xs,ys: Valuation;
942  eqn MergeOrderedValuations([],[])       = [];
943      MergeOrderedValuations([],xs)       = xs;
944      MergeOrderedValuations(xs,[])       = xs;
945      MergeOrderedValuations(x |> xs, y |> ys) =
946        if(x <= y, x |> MergeOrderedValuations(xs, y |> ys),
947          y |> MergeOrderedValuations(x |> xs, ys));
948
949  % Function that subtracts a valuation from another valuation.
950  map ValuationMinusValuationOrdered: Valuation#Valuation -> Valuation;
951  var x,y: Argument;
952      xs: Valuation;
953      ys: Valuation;
954  eqn ValuationMinusValuationOrdered([], ys) = [];
955      ValuationMinusValuationOrdered(xs, []) = xs;
956      ValuationMinusValuationOrdered(x |> xs, y |> ys) =
957        if(x == y, ValuationMinusValuationOrdered(xs, ys),
958          if(x < y, x |> ValuationMinusValuationOrdered(xs, y |> ys),
959              ValuationMinusValuationOrdered(x |> xs, ys)));
960
961  % Function that preserves all arguments in a valuation, for which the
962  % variables do not occur as a left hand side variable in a list
963  % of process parameters.
964  map RemoveArgumentWithDuplicateVariable: List(PP)#Valuation -> Valuation;
```

```
965      RemoveArgumentWithDuplicateVariable': List(PP)#Valuation -> Valuation;
966      OrderPP: List(PP) -> List(PP);
967      InsertPP: PP # List(PP) -> List(PP);
968  var p,q: PP;
969      lp,lq: List(PP);
970      v: Argument;
971      vl: Valuation;
972  eqn OrderPP([]) = [];
973      OrderPP(p |> lp) = InsertPP(p, OrderPP(lp));
974      InsertPP(p, []) = [p];
975      InsertPP(p, q |> lq)= if(p <=q, p |> q |> lq, q |> InsertPP(p, lq));
976      RemoveArgumentWithDuplicateVariable(lp, vl) =
977      RemoveArgumentWithDuplicateVariable'(OrderPP(lp), vl);
978      RemoveArgumentWithDuplicateVariable'([], vl) = vl;
979      RemoveArgumentWithDuplicateVariable'(lp, []) = [];
980      RemoveArgumentWithDuplicateVariable'(p |> lp, v |> vl) =
981        if(variable(p) == variable(v),
982          RemoveArgumentWithDuplicateVariable'(lp, vl),
983          if(variable(p) > variable(v),
984           v |> RemoveArgumentWithDuplicateVariable'(p |> lp, vl),
985           RemoveArgumentWithDuplicateVariable'(lp, v |> vl)));
986
987  %%%%%%%%%%%%%
988  %% Actions %%
989  %%%%%%%%%%%%%
990
991  % Transition relation function.
992  act a: List(ActionSemantic);
993
994  %%%%%%%%%%%%%%%
995  %% Processes %%
996  %%%%%%%%%%%%%%%
997
998  % Linear Process Equation.
999  proc X(p: ProcessTerm, s: Valuation) =
1000        sum r: ActionTransition.(r in R(p, s))-> a(pi_ac(r)).
1001          X(pi_t(r), pi_sigma'(r));
```

B.5.2 Model Specific Semantics

The model specific semantics describe the semantics that are specific for a set of models. Within this semantics we describe the allowed actions, variables, and (user defined) sorts, and the system of process equations.

```
1  % Sort for variables
2  sort Variable = struct bool(variablelabel: VariableLabel)?is_bool
3                       | nat(variablelabel: VariableLabel)?is_nat;
4
5  % Sort for values
6  sort Value   = struct bot | bool'(b: Bool)?is_bool | nat'(n: Nat)?is_nat;
7
8  % Sort for process equation labels
9  sort ProcessLabel = struct p0 | p1 | p2 | p3 | p4 | p5 | p6 | p7 | p8 | P | Q;
10
11 % Sort for action labels
12 sort ActionLabel = struct a | b | a1 | a2 | a3 | a4 | int;
13
14 % Sort for variable labels.
15 % Note that d'(Nat) may only be used to generate fresh variables
16 sort VariableLabel = struct v| v1 | v2 | v3 | d'(id: Nat)?is_d';
17
18 % Process Equations
```

```
19  map Def: ProcessLabel -> ProcessTerm;
20  eqn Def(p0) = Alpha([Act(a1, [de_var(bool(v1))])]);
21      Def(p1) = Seq(Alpha([Act(a1, [])]),
22                    Par(Alpha([Act(a2, [])]), Def(p1, [])));
23      Def(p2) = Seq(Alpha([Act(a1, [de_var(bool(v1))])]),
24                    Seq(Def(p3, [pp(bool(v1), de_val(bool'(false)))]),
25                      Alpha([Act(a1, [de_var(bool(v1))])])));
26      Def(p3) = Alpha([Act(a2, [de_var(bool(v1))])]);
27      Def(p4) = Seq(Alpha([Act(a2, [de_var(bool(v1))])]),
28                    Alpha([Act(a2, [de_var(bool(v1))])]));
29      Def(p5) = Seq(Alpha([Act(a1, [de_var(bool(v1))])]),
30                    Seq(Def(p4, [pp(bool(v1), de_val(bool'(false)))]),
31                      Alpha([Act(a1, [de_var(bool(v1))])])));
32      Def(p6) = Cond1(de_var(bool(v1)), Alpha([Act(a3, [de_var(bool(v1))])]));
33      Def(p7) = Seq(Alpha([Act(a1, [de_var(bool(v1))])]),
34                    Def(p7,
35                      [pp(bool(v1), de_expr_1(bool_op(neg), de_var(bool(v1))))]));
36      Def(p8) = Seq(Alpha([Act(a1, [de_var(bool(v1))])]),
37                    Alpha([Act(a2, [de_var(bool(v1))])]));
38      Def(P) = Sum(bool(v),  Seq(Alpha([ Act(a, [de_var(bool(v))])]),
39                    Alpha([Act(b, [ de_var(bool(v)) ])])));
40      Def(Q) = Alt(Seq(Alpha([ Act(a, [de_val(bool'(true))])]),
41                     Alpha([Act(b, [ de_val(bool'(true)) ])])),
42                   Seq(Alpha([ Act(a, [de_val(bool'(false))])]),
43                     Alpha([Act(b, [ de_val(bool'(false)) ])]));
44
45  % Restrict the selection of a value to a Particular domain
46  map RestrictDomain: Variable#Value   -> Bool;
47  var f: Func;
48      v1, v2: Value;
49      v: Variable;
50      w: Value;
51      vs: List(Variable);
52      ws: List(Value);
53  eqn RestrictDomain(v, w) =
54        (is_bool(v) && is_bool(w)) || (is_nat(v) && is_nat(w));
55
56  %% Generate a fresh variable with an appropriate sort
57  map GenFreshVar: Variable#Nat -> Variable;
58  var l: VariableLabel;
59      vid: Nat;
60  eqn GenFreshVar(bool(l), vid) = bool(d'(vid));
61      GenFreshVar(nat(l), vid)  = nat(d'(vid));
62
63
64  % Sorts for operators
65  sort Operator = struct neg | and | or | eq ;
66  sort Func     = struct bool_op(op: Operator)?is_bool
67                       | nat_op(op: Operator)?is_nat;
68
69  % Function to interpret meta notation data expression into values
70  map Sem_Dex: DataExpression#Valuation -> Value;
71  var d: Variable;
72      dvl: Value;
73      expr1: DataExpression;
74      expr2: DataExpression;
75      sigma: Valuation;
76      arg  : Argument;
77  eqn Sem_Dex(de_var(d), []) = bot;
78      Sem_Dex(de_var(d), arg |> sigma) =
79        if(variable(arg) == d, valvalue(arg), Sem_Dex(de_var(d), sigma));
80      Sem_Dex(de_val(dvl), sigma) = dvl;
81      Sem_Dex(de_expr_1(bool_op(neg), expr1), sigma) =
82        bool'(!Cast2InternalBool(Sem_Dex(expr1, sigma)));
83      Sem_Dex(de_expr_2(bool_op(and), expr1, expr2), sigma) =
```

```
84          bool'(Cast2InternalBool(Sem_Dex(expr1, sigma)) &&
85            Cast2InternalBool(Sem_Dex(expr2, sigma)));
86        Sem_Dex(de_expr_2(bool_op(or), expr1, expr2), sigma) =
87          bool'(Cast2InternalBool(Sem_Dex(expr1, sigma)) ||
88            Cast2InternalBool(Sem_Dex(expr2, sigma)));
89        Sem_Dex(de_expr_2(bool_op(eq), expr1, expr2), sigma) =
90          bool'(Cast2InternalBool(Sem_Dex(expr1, sigma)) ==
91            Cast2InternalBool(Sem_Dex(expr2, sigma)));
92
93    % Function to cast meta data expressions to mcrl2 data expressions
94    map Cast2InternalBool: Value -> Bool;
95        Cast2InternalNat: Value -> Nat;
96    var b: Bool;
97        n: Nat;
98    eqn Cast2InternalBool(bool'(b)) = b;
99        Cast2InternalNat(nat'(n))   = n;
100
101   % Function to substitute variables in data expressions
102   map VariableSubstitutionInDataExpression:
103         (Variable -> Variable)#DataExpression -> DataExpression;
104   var VarRename: Variable -> Variable;
105       value: Value;
106       v: Variable;
107       f: Func;
108       expr1, expr2: DataExpression;
109   eqn VariableSubstitutionInDataExpression(VarRename, de_val(value))=
110         de_val(value);
111       VariableSubstitutionInDataExpression(VarRename, de_var(v)) =
112         de_var(VarRename(v));
113       VariableSubstitutionInDataExpression(VarRename, de_expr_1(f, expr1))=
114         de_expr_1(f, VariableSubstitutionInDataExpression(VarRename, expr1));
115       VariableSubstitutionInDataExpression(VarRename, de_expr_2(f, expr1, expr2))=
116         de_expr_2(f, VariableSubstitutionInDataExpression(VarRename, expr1),
117           VariableSubstitutionInDataExpression(VarRename, expr2));
```

B.5.3 Input Models

The different input models are described here. Every *init* represents a different mCRL2 model in meta notation. The LPE that is used to generate the transitions carries the process label X. The LPE contains two arguments. The first argument denotes the mCRL2 model written in the meta notation. The second argument denotes the initial valuation for the model.

```
1   % Multi action tests
2   init X(alpha([Act(a1, [])]), []);
3   init X(alpha([]), []);
4   init X(alpha([ActionTau]), []);
5   init X(alpha([Act(a1, []), ActionTau]), []);
6   init X(alpha([Act(a1, [de_var(bool(v1))])]), [field(bool(v1), bool'(true))]);
7
8   % Alternative composition tests
9   init X(alt(alpha([Act(a1, [])]), deadlock), []);
10  init X(alt(alpha([Act(a1, [])]), alpha([Act(a2, [])])), []);
11
12  % Sequential composition tests
13  init X(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])), []);
14  init X(seq(alpha([Act(a1, [])]),
15         eq(alpha([Act(a2, [])]), alpha([Act(a2, [])]))), []);
16  init X(seq(seq(alpha([Act(a1, [de_var(bool(v1))])]),
17               alpha([Act(a2, [de_var(bool(v1))])])),
18           alpha([Act(a3, [de_var(bool(v1))])])),
```

```
19          [field(bool(v1), bool'(true))]);
20  init X(seq(alpha([Act(a1, [])]), seq(alpha([Act(a2, [])]),
21          alpha([Act(a3, [])]))), [field(bool(v1), bool'(true))]);
22  init X(alt(alpha([]), seq(alpha([Act(a1, [])]), deadlock)), []);
23
24  % Sum tests
25  init X(Sum(bool(v1), alpha([Act(a3, [de_var(bool(v1))])])), []);
26  init X(Sum(bool(v1), seq(alpha([Act(a3, [de_var(bool(v1))])]), deadlock)), []);
27  init X(Sum(bool(v1), seq(alpha([Act(a3, [de_var(bool(v1))])]), checkmark)), []);
28  init X(Sum(bool(v1), seq(alpha([Act(a3, [de_var(bool(v1))])]),
29         alpha([Act(a3, [])]))), []);
30  init X(Sum(bool(v1), seq(alpha([Act(a3, [de_var(bool(v1))])]),
31         alpha([Act(a3, [de_var(bool(v1))])])))), []);
32  init X(Sum(bool(v1), seq(Sum(bool(v1),
33         alpha([Act(a1, [de_var(bool(v1))])])),
34         alpha([Act(a3, [de_var(bool(v1))])])))), []);
35
36  % Condition tests
37  init X(cond1(de_var(bool(v1)), alpha([Act(a3, [de_var(bool(v1))])])),
38         [field(bool(v1), bool'(true))]);
39  init X(cond1(de_var(bool(v1)), alpha([Act(a3, [de_var(bool(v1))])])),
40         [field(bool(v1), bool'(true))]);
41  init X(cond1(de_var(bool(v1)), alpha([Act(a3, [de_var(bool(v1))])])),
42         [field(bool(v1), bool'(false))]);
43  init X(cond1(de_expr_1(bool_op(neg), de_var(bool(v1))),
44         alpha([Act(a3, [de_var(bool(v1))])])),
45           [field(bool(v1), bool'(false))]);
46  init X(cond2(de_var(bool(v1)), alpha([Act(a1, [de_var(bool(v1))])]),
47         alpha([Act(a2, [de_var(bool(v1))])])),
48           [field(bool(v1), bool'(true))]);
49  init X(cond2(de_var(bool(v1)), alpha([Act(a1, [de_var(bool(v1))])]),
50         alpha([Act(a2, [de_var(bool(v1))])])),
51           [field(bool(v1), bool'(false))]);
52  init X(Sum(bool(v1), cond1(de_var(bool(v1)),
53         alpha([Act(a3, [de_var(bool(v1))])])))), []);
54  init X(Sum(bool(v1), cond2(de_var(bool(v1)),
55         alpha([Act(a1, [de_var(bool(v1))])]),
56           alpha([Act(a3, [de_var(bool(v1))])])))), []);
57  init X(Sum(bool(v1), cond2(de_expr_1(bool_op(neg), de_var(bool(v1))),
58         alpha([Act(a1, [de_var(bool(v1))])]),
59           alpha([Act(a3, [de_var(bool(v1))])])))), []);
60
61  % Parallel tests
62  init X(par(alpha([Act(a1, [])]), alpha([Act(a2, [])])), []);
63  init X(par(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])),
64         seq(alpha([Act(a3, [])]), alpha([Act(a4, [])])))), []);
65
66  % Sync tests
67  init X(sync(alpha([Act(a1, [])]), alpha([Act(a2, [])])), []);
68  init X(sync(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])), alpha([Act(a3, [])])), []);
69  init X(sync(alpha([Act(a1, [])]), seq(alpha([Act(a2, [])]), alpha([Act(a3, [])])))), []);
70  init X(sync(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])),
71          seq(alpha([Act(a3, [])]), alpha([Act(a4, [])])))), []);
72
73  % Left merge tests
74  init X(lmerge(alpha([Act(a1, [])]), alpha([Act(a2, [])])), []);
75  init X(lmerge(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])), alpha([Act(a3, [])])), []);
76  init X(lmerge(alpha([Act(a1, [])]), seq(alpha([Act(a2, [])]), alpha([Act(a3, [])])))), []);
77  init X(lmerge(seq(alpha([Act(a1, [])]), alpha([Act(a2, [])])),
78          seq(alpha([Act(a3, [])]), alpha([Act(a4, [])])))), []);
79
80  % Allow tests
81  init X(Allow({}, alpha([Act(a1, [])])), []);
82  init X(Allow({{a2: 1}}, alpha([Act(a1, [])])), []);
83  init X(Allow({{a1: 1}}, alpha([Act(a1, [])])), []);
```

```
84  init X(Allow({{a1: 1}}, seq(alpha([Act(a1, [])]), alpha([Act(a1, [])]))), []);
85  init X(Allow({{a2: 1}}, seq(alpha([Act(a1, [])]), alpha([Act(a2, [])]))), []);
86  init X(Allow({{a1: 1}}, seq(alpha([Act(a1, [])]), alpha([Act(a2, [])]))), []);
87
88  % Block tests
89  init X(Block({a1}, alpha([Act(a1, [])])), []);
90  init X(Block({a2}, alpha([Act(a1, [])])), []);
91  init X(Block({a2}, seq(alpha([Act(a1, [])]), alpha([Act(a2, [])]))), []);
92  init X(Block({a2}, alpha([Act(a1, []), Act(a2, [])])), []);
93
94  % Action rename tests
95  init X(Rename(ID[a2->a1], alpha([Act(a2, [])])), []);
96  init X(Rename(ID[a2->a1], seq(alpha([Act(a2, [])]), alpha([Act(a2, [])]))), []);
97
98  % Prehide tests
99  init X(Prehide({a2}, alpha([Act(a2, [])])), []);
100 init X(Prehide({a2}, seq(alpha([Act(a2, [])]), alpha([Act(a1, [])]))), []);
101
102 % Hide tests
103 init X(Hide({a2}, alpha([Act(a2, [])])), []);
104 init X(Hide({a2}, seq(alpha([Act(a2, [])]), alpha([Act(a1, [])]))), []);
105
106 % Communication tests
107 init X(Comm([communication([a2, a2], a1)],
108         alpha([Act(a2, []), Act(a1, []), Act(a2, [])])), []);
109 init X(Comm([communication([a2, a2], a1)],
110         alpha([Act(a2, []), Act(a1, []), Act(a2, [de_var(bool(v1))])])),
111       [field(bool(v1), bool'(true))]);
112
113 % Process Equation tests
114 init X(Def(p0, [pp(bool(v1), de_val(bool'(false)))]), [field(bool(v1), bool'(true))]);
115 init X(Def(p0, [pp(bool(v1), de_var(bool(v2)))]), [field(bool(v1), bool'(true))]);
116 init X(Def(p0, [pp(bool(v2), de_var(bool(v1)))]), [field(bool(v1), bool'(true))]);
117 init X(Def(p2, []), [field(bool(v1), bool'(true))]);
118 init X(Def(p5, []), [field(bool(v1), bool'(true))]);
119 init X(Def(p4, [pp(bool(v1), de_val(bool'(false)))]), [field(bool(v1), bool'(true))]);
120 init X(seq(Def(p4, [pp(bool(v1), de_val(bool'(false)))]),
121        alpha([Act(a2, [de_var(bool(v1))])])), [field(bool(v1), bool'(true))]);
122 init X(Def(p6, [pp(bool(v1), de_val(bool'(true)))]),
123        [field(bool(v1), bool'(false))]);
124 init X(Sum(bool(v1), Sum(bool(v2),
125        alpha([Act(a1, [de_var(bool(v1)), de_var(bool(v2))])]))), []);
126 init X(par(Def(p0, [pp(bool(v1), de_val(bool'(false)))]),
127         Def(p0, [pp(bool(v1), de_val(bool'(true)))])), []);
128 init X(par(Def(p8, [pp(bool(v1), de_val(bool'(true)))]),
129         Def(p8, [pp(bool(v1), de_val(bool'(true)))])), []);
130 init X(par(Def(p8, [pp(bool(v1), de_val(bool'(false)))]),
131        Def(p8, [pp(bool(v1), de_val(bool'(true)))])), []);
132 init X(Def(P, []), []);
133 init X(Def(Q, []), []);
134 init X(par(Def(P, []), Def(P, [])), []);
135 init X(par(Def(Q, []), Def(Q, [])), []);
```

Bibliography

[65A10] 65A - System aspects. Functional safety of electrical/electronic/programmable electronic safety-related systems - S+ IEC 61508 (Edition 2.0). CD, 4 2010.

[ABE10] M.F. van Amstel, M.G.J. van den Brand, and L.J.P. Engelen. An exercise in iterative domain-specific language design. In A. Capiluppi, A. Cleve, and N. Moha, editors, *EVOL/IWPSE*, pages 48–57. ACM, 2010.

[ABPV08] 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 A. Vallecillo, J. Gray, and A. Pierantonio, editors, *ICMT*, volume 5063 of *LNCS*, pages 61–75. Springer, 2008.

[ABV94] L. Aceto, B. Bloom, and F.W. Vaandrager. Turning SOS Rules into Equations. *Inf. Comput.*, 111(1):1–52, 1994.

[AFV01] L. Aceto, W.J. Fokkink, and C. Verhoef. Conservative Extension in Structural Operational Semantics. In *Current Trends in Theoretical Computer Science*, pages 504–524. World Scientific, 2001.

[AH89] L. Aceto and M. Hennessy. Termination, Deadlock and Divergence. In M.G. Main, A. Melton, M.W. Mislove, and D.A. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 442 of *LNCS*, pages 301–318. Springer, 1989.

[AIMR09] L. Aceto, A. Ingolfsdottir, M.R. Mousavi, and M.A. Reniers. Algebraic Properties for Free! *Bulletin of the EATCS*, 99:81–104, October 2009.

[Are02] D.B. Aredo. A Framework for Semantics of UML Sequence Diagrams in PVS. *Journal of Universal Computer Science*, 8(7):674–697, 2002.

[Aut] AutoDesk. 3D Studio Max 2009. `http://usa.autodesk.com`. Visited: September 25, 2012.

[AvdBE11] 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 F. Durán and V. Rusu, editors, *AMMSE*, volume 56 of *EPTCS*, pages 65–79, 2011.

[AvdBEV12] S. Andova, M.G.J. van den Brand, L.J.P. Engelen, and T. Verhoef. MDE Basics with a DSL Focus. In Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio, editors, *Formal Methods for Model-Driven Engineering*, volume 7320 of *LNCS*, pages 21–57. Springer Berlin / Heidelberg, 2012.

[AW02] T. Archer and A. Whitechapel. *Inside C*. Pro-Developer Series. Microsoft Press, second edition edition, 2002.

[BB88] J.C.M. Baeten and J.A. Bergstra. Processen en procesexpressies (Dutch). *Informatie*, 30(3):177–248, 1988.

[BB97] J.M.C. Baeten and J.A. Bergstra. Discrete Time Process Algebra: Absolute Time, Relative Time and Parametric Time. *Fundam. Inform.*, 29(1-2):51–76, 1997.

[BBG97] M. Bezem, R.N. Bol, and J.F. Groote. Formalizing Process Algebraic Verifications in the Calculus of Constructions. *Formal Asp. Comput.*, 9(1):1–48, 1997.

[BBM03] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BBNS09] S. Bensalem, M. Bozga, T-H Nguyen, and J. Sifakis. D-finder: A tool for compositional deadlock detection and verification. In A. Bouajjani and O. Maler, editors, *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009), Grenoble, France*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[BBR09] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes (Cambridge Tracts in Theoretical Computer Science)*. Cambridge University Press, December 2009.

[BCN+09] D. A. van Beek, P. Collins, D. E. Nadales, J. E. Rooda, and R. R. H. Schiffelers. New Concepts in the Abstract Format of the Compositional Interchange Format. In A. Giua, C. Mahuela, M. Silva, and J. Zaytoon, editors, *3rd IFAC Conference on Analysis and Design of Hybrid Systems*, pages 250–255, Zaragoza, Spain, 2009.

[BCR00] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa,*

USA, May 20-27, 2000, Proceedings, volume 1816 of *LNCS*, pages 293–308. Springer, 2000.

[BDH01] D. Bosnacki, D. Dams, and L. Holenderski. A Heuristic for Symmetry Reductions with Scalarsets. In J.N. Oliveira and P. Zave, editors, *FME*, volume 2021 of *LNCS*, pages 518–533. Springer, 2001.

[BFG+01] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. Computer Aided Verification (CAV 2001)*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[BG96] R.N. Bol and J.F. Groote. The Meaning of Negative Premises in Transition System Specifications. *J. ACM*, 43(5):863–914, 1996.

[BGR+03] D.A. van Beek, Niek G.J., J.E. Rooda, R.R.H. Schiffelers, K.L. Man, and M.A. Reniers. Relating Chi to hybrid automata. In S.E. Chick, P.J. Sánchez, D.M. Ferrin, and D.J. Morrice, editors, *Proceedings of the 2003 Winter Simulation Conference*, pages 632–640, 2003.

[BHBM07] J.M.J. Beckers, W.P.M.H. Heemels, B.H.M. Bukkems, and G.J. Muller. Effective industrial modeling for high-tech systems: The example of Happy Flow. In *Seventeenth Annual International Symposium of the International Council On Systems Engineering (INCOSE)*, 2007.

[BHMM00] C. de O. Braga, E.H. Haeusler, J. Meseguer, and P.D. Mosses. Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic. In *AMAST*, pages 407–421, 2000.

[BHMM02] C. de O. Braga, E.H. Haeusler, J. Meseguer, and P.D. Mosses. Mapping Modular SOS to Rewriting Logic. In *LOPSTR*, pages 262–277, 2002.

[BHR+08] 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 1, Eindhoven University of Technology, 2008.

[BK84] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109–137, 1984.

[BK85] J.A. Bergstra and J.W. Klop. Algebra of Communicating Processes with Abstraction. *Theor. Comput. Sci.*, 37:77–121, 1985.

[BLL+95] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in LNCS, pages 232–243. Springer, October 1995.

[Blu05] Bluespec. *Automatic Generation of Control Logic with Bluespec SystemVerilog*, Februari 2005. Available at: `http://www.bluespec.com/forum/download.php?id=63`.

[BM02] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monographs. Springer-Verlag, Berlin, Germany, 2002.

[BM05] C. Braga and J. Meseguer. Modular Rewriting Semantics in Practice. *ENTCS*, 117:393–416, 2005.

[BMR+05] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and semantics of timed Chi. Computer Science Report 05-09, Technische Universiteit Eindhoven, March 2005.

[BMR+06] D.A. van Beek, K.L. Man, M.A. Reniers, J.E. Rooda, and R.R.H. Schiffelers. Syntax and consistent equation semantics of hybrid chi. *J. Log. Algebr. Program.*, 68(1-2):129–210, 2006.

[Bör98] E. Börger. High Level System Design and Analysis Using Abstract State Machines. In *FM-Trends*, volume 1641 of *LNCS*, pages 1–43. Springer, 1998.

[BPSM+08] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[BPW93] J.A. Bergstra, A. Ponse, and J. van Wamel. Process Algebra with Backtracking. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *REX School/Symposium*, volume 803 of *LNCS*, pages 46–91. Springer, 1993.

[BR00] T. Ball and S.K.i Rajaman. Bebop: A Symbolic Model Checker for Boolean Programs. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification (SPIN'00), Stanford, CA, USA*, volume 1885 of *LNCS*, pages 113–130. Springer, 2000.

[BR01] T. Ball and S.K.i Rajaman. Automatically validating temporal safety properties of interfaces. In M.B. Dwyer, editor, *Proceedings of the 8th international SPIN workshop on Model Checking Software (SPIN'01), Toronto, Ontario, Canada*, volume 2057 of *LNCS*, pages 103–122. Springer, 2001.

[BR04] J.M.C. Baeten and M.A. Reniers. Timed Process Algebra (With a Focus on Explicit Termination and Relative-Timing). In M. Bernardo and F. Corradini, editors, *SFM*, volume 3185 of *LNCS*, pages 59–97. Springer, 2004.

[Bra92] J.C. Bradfield. *Verifying Temporal Properties of Systems*. Progress in Theoretical Computer Science. Birkhäuser, 1992.

[Bra01] C. de O. Braga. *Rewriting Logic as a Semantic Framework for Modular Structural Operational Semantics*. PhD thesis, Pontifícia Universidade Católica do Rio de Janeiro, 2001.

[BRdSV89] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process Calculi, from Theory to Practice: Verification Tools. In *Automatic Verification Methods for Finite State Systems*, pages 1–10, 1989.

[BRSR07] D.A. van Beek, M.A. Reniers, R.R.H. Schiffelers, and J.E. Rooda. Foundations of a Compositional Interchange Format for Hybrid Systems. In A. Bemporad, A. Bicchi, and G.C. Buttazzo, editors, *HSCC*, volume 4416 of *LNCS*, pages 587–600. Springer, 2007.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12:260–261, May 1969.

[BTW+05] E. Bortnik, N. Trčka, A.J. Wijs, B. Luttik, J.M. van de Mortel-Fronczak, J.C.M. Baeten, W.J. Fokkink, and J.E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and UPPAAL. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[But92] K-H. Buth. Using SOS Definitions in Term Rewriting Proofs. In *Larch*, Workshops in Computing, pages 36–54. Springer, 1992.

[But94] K-H. Buth. Simulation of SOS Definitions with Term Rewriting Systems. In *ESOP*, volume 788 of *LNCS*, pages 150–164. Springer, 1994.

[BV93] J.C.M. Baeten and C. Verhoef. A Congruence Theorem for Structured Operational Semantics with Predicates. In *CONCUR*, pages 477–492, 1993.

[BV07] C. Braga and A. Verdejo. Modular Structural Operational Semantics with Strategies. *ENTCS*, 175(1):3–17, 2007.

[BvDH+01] M.G.J Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In R. Wilhelm, editor, *CC*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

[CCGT09] B. Combemale, X. Crégut, P-L. Garoche, and X. Thirioux. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *JSW*, 4(9):943–958, 2009.

[CCR08] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL Class Diagrams using Constraint Programming. In *ICSTW '08*, pages 73–80. IEEE Computer Society, 2008.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Lauback, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering (ICSE 2000), Limerick, Ireland*, pages 439–448, 2000.

[CGJ+03] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, 1999.

[CH90] R. Cleaveland and M. Hennessy. Priorities in Process Algebras. *Inf. Comput.*, 87(1/2):58–77, 1990.

[Chu32] A. Church. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics*, 2, 1932.

[CMS95] R. Cleaveland, E. Madelaine, and S. Sims. A Front-End Generator for Verification Tools. In *TACAS*, volume 1019 of *LNCS*, pages 153–173. Springer, 1995.

[CS96] R. Cleaveland and S. Sims. The NCSU Concurrency Workbench. In *CAV*, volume 1102 of *LNCS*, pages 394–397. Springer, 1996.

[CW96] E.M. Clarke and J.M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[CW02] T. Clark and J. Warmer, editors. *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, volume 2263 of *LNCS*. Springer, 2002.

[Dar88] I.F. Darwin. *Checking C programs with Lint*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1988.

[dBIM06] M.G.J. den Brand, J. Iversen, and P.D. Mosses. An Action Environment. *Sci. Comput. Program.*, 61(3):245–264, 2006.

[dBvL80] J.W. de Bakker and J. van Leeuwen, editors. *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *LNCS*. Springer, 1980.

[Del11] T. Delissen. Design and Validation of a Model-Driven Engineering Environment for the Specification and Transformation of T-ReCS models. Master's thesis, Technische Universiteit Eindhoven, 2011.

[DGP02] P. Degano, F. Gadducci, and C. Priami. A Causal Semantics for CCS via Rewriting Logic. *Theor. Comput. Sci.*, 275(1-2):259–282, 2002.

[DMY02] A. David, M.O. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In *FASE*, volume 2306 of *LNCS*, pages 218–232. Springer, 2002.

[DRJK+06] D. Di Ruscio, F. Jouault, I. Kurtev, J. Bézivin, and A. Pierantonio. Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report n. 06.02, Laboratoire d'Informatique de Nantes-Atlantique, April 2006.

[dS85] R. de Simone. Higher-Level Synchronising Devices in Meije-SCCS. *Theor. Comput. Sci.*, 37:245–267, 1985.

[DS09] E.G. Daylight and S.K. Shukla. On the Difficulties of Concurrent-System Design, Illustrated with a 2×2 Switch Case Study. In *FM*, volume 5850 of *LNCS*, pages 273–288. Springer, 2009.

[EC80] E.A. Emerson and E.M. Clarke. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In de Bakker and van Leeuwen [dBvL80], pages 169–181.

[EM02] S. Eker and A. Meseguer, J.and Sridharanarayanan. The Maude LTL Model Checker. *ENTCS*, 71:162–187, 2002.

[EW05] J. Evermann and Y. Wand. Toward Formalizing Domain Modeling Semantics in Language Syntax. *IEEE Trans. Software Eng.*, 31(1):21–37, 2005.

[Fok07] W.J. Fokkink. *Modelling Distributed Systems*. Springer Berlin Heidelberg, 2007.

[FV98] W.J. Fokkink and C. Verhoef. A Conservative Look at Operational Semantics with Variable Binding. *Inf. Comput.*, 146(1):24–54, 1998.

[Gal03] V. Galpin. A format for semantic equivalence comparison. *Theor. Comput. Sci.*, 309(1-3):65–109, 2003.

[GG89] R.J. van Glabbeek and U. Goltz. Equivalence Notions for Concurrent Systems and Refinement of Actions (Extended Abstract). In A. Kreczmar and G. Mirkowska, editors, *Proceedings of Mathematical Foundations of Computer Science 1989 (MFCS'89), Porabka-Kozubnik, Poland*, volume 379 of *LNCS*, pages 237–248. Springer, 1989.

[GH93] J.V. Guttag and J.J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GKM+08] J.F. Groote, J.J.A Keiren, A.H.J. Mathijssen, B. Ploeger, F.P.M. Stappers, C. Tankink, Y.S. Usenko, M.J. van den Weerdenburg, W. Wesselink, T.A.C. Willemse, and J. van der Wulp. The mCRL2 toolset. In *Proceedings International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008)*, 2008.

[GKOT00] Y. Gurevich, P.W. Kutter, M. Odersky, and L. Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *LNCS*. Springer, 2000.

[GKS+11] J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, W. Wesselink, and T.A.C. Willemse. Experiences in developing the mCRL2 toolset. *Softw., Pract. Exper.*, 41(2):143–153, 2011.

[GM99] J.F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In A.M. Haeberer, editor, *Proc. Algebraic Methodology And Software Technology (AMAST 1998)*, volume 1548 of *LNCS*, pages 74–90. Springer, 1999.

[GMR+06] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van den Weerdenburg. The Formal Specification Language mCRL2. In *MMOSS*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[GMR+09] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van den Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, chapter 4, pages 99–128. Taylor & Francis Group, 2009.

[GMWU06] J.F. Groote, A.H.J. Mathijssen, M. van Weerdenburg, and Y.S. Usenko. From μCRL to mCRL2: Motivation and Outline. *ENTCS*, 162:191–196, 2006.

[God97] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 174–186. ACM Press, 1997.

[GP93] J.F. Groote and A. Ponse. Proof Theory for μCRL: A Language for Processes with Data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages*, Workshops in Computing, pages 232–251. Springer, 1993.

[GPU01] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *J. Log. Algebr. Program.*, 48(1-2):39–70, 2001.

[GPW03] J.F. Groote, J. Pang, and A.G. Wouters. Analysis of a distributed system for lifting trucks. *J. Log. Algebr. Program.*, 55(1-2):21–56, 2003.

[GR01] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier Science Publishers B.V., Amsterdam, 2001.

[Gro93] J.F. Groote. Transition system specifications with negative premises. *Theor. Comput. Sci.*, 118(2):263–299, 1993.

[GV92] J.F. Groote and F.W. Vaandrager. Structured Operational Semantics and Bisimulation as a Congruence. *Inf. Comput.*, 100(2):202–260, 1992.

[GW05a] J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, 2005.

[GW05b] J.F. Groote and T.A.C. Willemse. Parameterised Boolean Equation Systems. *Theor. Comput. Sci.*, 343(3):332–369, 2005.

[HA00] J.C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 511–519, Piscataway, NJ, USA, 2000. IEEE Press.

[Haa] K. Haaland. JIT Software Development-Inside the Eclipse Software Development Process.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[Har99] P.H. Hartel. LETOS - a Lightweight Execution Tool for Operational Semantics. *Softw., Pract. Exper.*, 29(15):1379–1416, 1999.

[Har06] W. Harrison. Eating Your Own Dog Food. *IEEE Software*, 23(3):5–7, 2006.

[Hen96] T.A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[HHJW07] P. Hudak, J. Hughes, S.L.P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In B.G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

[HHVOM07] M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Using Maude and Its Strategies for Defining a Framework for Analyzing Eden Semantics. *ENTCS*, 174(10):119–137, 2007.

[HKW11] Y-L Hwong, V.J.J. Kusters, and T.A.C. Willemse. Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider. In F. Arbab and M. Sirjani, editors, *FSEN*, volume 7141 of *LNCS*, pages 174–189. Springer, 2011.

[HM80] M. Hennessy and R. Milner. On Observing Nondeterminism and Concurrency. In de Bakker and van Leeuwen [dBvL80], pages 299–309.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol01] G.J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD 2001), Newcastle upon Tyne, UK*, pages 3–10. IEEE Computer Society Press, 2001.

[Hor51] A. Horn. On Sentences Which are True of Direct Unions of Algebras. *J. Symb. Log.*, 16(1):14–21, 1951.

[HR04] M. Huth and M.D. Ryan. *Logic in computer science - Modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[HVB00] J. den Hartog, E.P. de Vink, and J.W. de Bakker. Metric Semantics and Full Abstractness for Action Refinement and Probabilistic Choice. *ENTCS*, 40:72–99, 2000.

[HWW01] F. van Ham, H. van de Wetering, and J.J. van Wijk. Visualization of State Transition Graphs. In *INFOVIS*, pages 59–63, 2001.

[IEE91] IEEE. IEEE Standard for the Scheme Programming Language. *IEEE Std 1178-1990*, 1991.

[Inc] PolySpace Inc. Polyspace verification toolsuite. `http://www.polyspace.com`. Visited: September 25, 2012.

[ISO05] OMG. *Meta Object Facility (MOF) Specification*, iso/iec 19502:2005 edition, 2005.

[Joh78] S.C. Johnson. Lint, a C program checker. Technical Report Comp. Sci. Tech. Rep. 65, Bell Laboratories, 1978.

[JRH+99] S.L.P. Jones, A. Reid, F. Henderson, C.A.R. Hoare, and S. Marlow. A Semantics for Imprecise Exceptions. In *PLDI*, pages 25–36, 1999.

[JS04] S. Jansamak and A. Surarerks. Formalization of UML statechart models using Concurrent Regular Expressions. In *ACSC '04*, pages 83–88, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[JS08] E.K. Jackson and W. Schulte. Model Generation for Horn Logic with Stratified Negation. In *FORTE*, volume 5048 of *LNCS*, pages 1–20. Springer, 2008.

[JS09] E.K. Jackson and J. Sztipanovits. Formalizing the Structural Semantics of Domain-Specific Modeling Languages. *Software and System Modeling*, 8(4):451–478, 2009.

[Kle09] A. Kleppe. *Software Language Engineering*. Addisson-Wesley, 2009.

[Kof07] J. Kofron. Checking software component behavior using behavior protocols and SPIN. In Y. Cho, R.L. Wainwright, H. Haddad, S.Y. Shin, and Y.W. Koo, editors, *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07), Seoul, Korea*, pages 1513–1517. ACM Press, 2007.

[Koz83] D. Kozen. Results on the Propositional mu-Calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[KR11] J.J.A. Keiren and M.A. Reniers. Type checking mCRL2. Technical Report 11, Technische Universiteit Eindhoven, 2011.

[Kus01] S. Kuske. A Formal Semantics of UML State Machines Based on Structured Graph Transformation. In *UML*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lam02] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[LME04] P. Leven, T. Mehler, and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In S. Graf and L. Mounier, editors, *Proceedings of the 11th International SPIN Workshop on Model Checking Software (SPIN), Barcelona, Spain*, volume 2989 of *LNCS*, pages 39–56. Springer, 2004.

[LN03] K.G. Larsen and P. Niebert, editors. *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, volume 2791 of *LNCS*. Springer, 2003.

[LS92] K.G. Larsen and A. Skou. Compositional Verification of Probabilistic Processes. In R. Cleaveland, editor, *CONCUR*, volume 630 of *LNCS*, pages 456–471. Springer, 1992.

[McL07] J. McLoone. Eating Your Own Dogfood, May 2007.

[Mes92] J. Meseguer. Conditioned Rewriting Logic as a United Model of Concurrency. *Theor. Comput. Sci.*, 96(1):73–155, 1992.

[MOM96] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. *ENTCS*, 4, 1996.

[Mon05] M. Monteban. Reduction Algorithms on Linear Process Equations. IS-TI 005, Vrije Universiteit Amsterdam, 2005.

[Mos04a] P.D. Mosses. Exploiting Labels in Structural Operational Semantics. In *SAC*, pages 1476–1481, 2004.

[Mos04b] P.D. Mosses. Modular Structural Operational Semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[MP07] A.H.J. Mathijssen and A.J. Pretorius. Verified Design of an Automated Parking Garage. In L. Brim, B.R. Haverkort, M. Leucker, and J.C. van de Pol, editors, *Proc. FMICS and PDMC 2006*, volume 4346 of *LNCS*, pages 165–180. Springer, 2007.

[MR06] M.R. Mousavi and M.A. Reniers. Prototyping SOS Meta-theory in Maude. *ENTCS*, 156(1):135–150, 2006.

[MRG07] M.R. Mousavi, M.A. Reniers, and J.F. Groote. SOS Formats and Meta-theory: 20 Years After. *Theor. Comput. Sci.*, 373(3):238–272, 2007.

[MSW09] M.G Meulen, F.P.M. Stappers, and T.A.C. Willemse. Breath-Bounded Model Checking. Computer Science Report No. 09-03, Eindhoven University of Technology, March 2009.

[MWW04] S. Mauw, W.T. Wiersma, and T.J.H. Willemse. Language-Driven System Design. *IJSEKE*, 14(6):625–663, 2004.

[Nie04] N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-reactive Scheduling*. PhD thesis, Technische University Eindhoven, 2004.

[NNH99] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[NO96] E. Najm and F. Olsen. Reactive EFSMs - reactive Promela/RSPIN. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings*, volume 1055 of *LNCS*, pages 349–368. Springer, 1996.

[ÖBM10] P.C Ölveczky, A. Boronat, and J. Meseguer. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude. In J. Hatcliff and E. Zucca, editors, *FMOODS/FORTE*, volume 6117 of *LNCS*, pages 47–62. Springer, 2010.

[OW10] S. Orzan and T.A.C. Willemse. Invariants for Parameterised Boolean Equation Systems. *Theor. Comput. Sci.*, 411(11-13):1338–1371, 2010.

[PL99] I. Paltor and J. Lilius. Formalising UML State Machines for Model Checking. In *UML*, volume 1723 of *LNCS*, pages 430–445. Springer, 1999.

[Plo04] G.D. Plotkin. A Structural Approach to Operational Semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[PRQ] PRQA. QA-C++ toolsuite. `http://www.programmingresearch.com/`. Visited: September 25, 2012.

[PT08] B. Ploeger and C. Tankink. Improving an Interactive Visualization of Transition Systems. In *Proceedings of the 4th ACM Symposium on Software Visualization 2008 (SoftVis 2008)*, pages 115–124. ACM, 2008.

[PW06] A.J. Pretorius and J.J. van Wijk. Visual Analysis of Multivariate State Transition Graphs. *IEEE Trans. Vis. Comput. Graph.*, 12(5):685–692, 2006.

[PW07] A.J. Pretorius and J.J. van Wijk. Bridging the semantic gap: Visualizing transition graphs with user-defined diagrams. *IEEE Computer Graphics and Applications*, 27(5):58–66, 2007.

[PWW11] B. Ploeger, W. Wesselink, and T.A.C. Willemse. Verification of reactive systems via instantiation of Parameterised Boolean Equation Systems. *Inf. Comput.*, 209(4):637–663, 2011.

[Rep93] J.H. Reppy. Concurrent ML: Design, Application and Semantics. In P.E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *LNCS*, pages 165–198. Springer, 1993.

[RGZW02] M.A. Reniers, J.F. Groote, M. van der Zwaag, and J. van Wamel. Completeness of Timed mCRL. *Fundam. Inform.*, 50(3-4):361–402, 2002.

[RJB04] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[Roo07] N. Roos. Océ geeft aanzet tot open innovatie in inkjet, August 2007. Mechatronica Magazine.

[RRH10] A. Riesco and J. Rodríguez-Hortalá. A Natural Implementation of Plural Semantics in Maude. *ENTCS*, 253(7):165–175, 2010.

[Sax94] R. Sax. *Classic Home Desserts: A Treasury of Heirloom and Contemporary Recipes from Around the World*. Houghton Mifflin Harcourt, 1994.

[Sco70] D.S. Scott. Outline of a Mathematical Theory of Computation. Technical Monograph PRG–2, Oxford University Computing Laboratory, Oxford, England, November 1970.

[Sofa] The Maude system. `http://maude.cs.uiuc.edu/`. Visited: September 25, 2012.

[Sofb] The mCRL2 toolset. `http://www.mcrl2.org/`. Visited: September 25, 2012.

[Sofc] Java PathFinder. `http://javapathfinder.sourceforge.net`. Visited: September 25, 2012.

[SR09] F.P.M. Stappers and M.A. Reniers. Verification of safety requirements for program code using data abstraction. *ECEASST*, 23, 2009.

[SRG09] F.P.M. Stappers, M.A. Reniers, and J.F. Groote. Suitability of mCRL2 for Concurrent-System Design: A 2×2 Switch Case Study. In *FMCO*, volume 6286 of *LNCS*, pages 166–185. Springer, 2009.

[SRG10] F.P.M. Stappers, M.A. Reniers, and J.F. Groote. Grip op Correcte Software (Dutch). In *Release*, volume 4, pages 18–21. Array Publications, 2010.

[SRGW11] F.P.M. Stappers, M.A. Reniers, J.F. Groote, and S. Weber. Dogfooding the Structural Operational Semantics of mCRL2. Computer Science Report No. 11-18, Eindhoven University of Technology, December 2011.

[SRW11a] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS Specifications to Linear Processes. Computer Science Report No. 11-07, Eindhoven University of Technology, May 2011.

[SRW11b] F.P.M. Stappers, M.A. Reniers, and S. Weber. Transforming SOS Specifications to Linear Processes. In *FMICS*, volume 6959 of *LNCS*, pages 196–211. Springer, 2011.

[SRWG12] F.P.M. Stappers, M.A. Reniers, S. Weber, and J.F. Groote. Dogfooding the Formal Semantics of mCRL2. In *SEW*, volume 35. IEEE, 2012. To Appear.

[SS08] G. Singh and S.K. Shukla. Verifying Compiler Based Refinement of Bluespec^{TM} Specifications Using the SPIN Model Checker. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, volume 5156 of *LNCS*, pages 250–269, Berlin, Heidelberg, 2008. Springer-Verlag.

[SSR08a] F.P.M. Stappers, L.J.A.M. Somers, and M.A. Reniers. Multidisciplinary Modelling: Current status and expectations in the Dutch TWINS consortium. In *ICSSEA08*, pages S5.2:1–10, 2008.

[SSR08b] F.P.M. Stappers, L.J.A.M. Somers, and M.A. Reniers. Multidisciplinary Modelling in the Netherlands. In J. Heidrich and D. Falessi, editors, *PROFES*, volume Short Paper Session Proceedings, pages 25–28, 2008.

[SSR09] F.P.M. Stappers, L.J.A.M. Somers, and M.A. Reniers. La modélisation multidisciplinaire: État d'avancements et attentes du projet Neérlandais TWINS. *Génie Logiciel*, 88:26–35, March 2009.

[SW09] D.A. Sadilek and G. Wachsmuth. Using Grammarware Languages to Define Operational Semantics of Modelled Languages. In *TOOLS (47)*, volume 33 of *LNBIP*, pages 348–356. Springer, 2009.

[SWR+11a] F.P.M. Stappers, S. Weber, M.A. Reniers, S. Andova, and I. Nagy. Formalizing a Domain Specific Language Using SOS: An Industrial Case Study. In A.M. Sloane and U. Aßmann, editors, *SLE*, volume 6940 of *LNCS*, pages 223–242. Springer, 2011.

[SWR+11b] F.P.M. Stappers, S. Weber, M.A. Reniers, S. Andova, and I. Nagy. Formalizing a Domain Specific Language Using SOS: An Industrial Case Study. In U. Assmann, J. Saraiva, and A. Sloane, editors, *SLE*, LNCS, Pre-Proceedings, pages 223–242, July 2011.

[TDP03] O. Tkachuk, M.B. Dwyer, and C.S. Pasareanu. Automated Environment Generation for Software Model Checking. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada*, pages 116–129. IEEE Computer Society Press, 2003.

[Ter97] P.D. Terry. *Compilers and Compiler Generators: An Introduction with C++*. Coriolis Group, March 1997.

[Ton98] H. Tonino. A Sound and Complete SOS-Semantics for Non-Distributed Deterministic Abstract State Machines. In *Workshop on Abstract State Machines*, pages 91–110, 1998.

[Tur85] D.A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *FPCA*, pages 1–16, 1985.

[Use02] Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Eindhoven University of Technology, December 2002.

[vdPV97] P.H.A van der Putten and J.P.M. Voeten. *Specification of reactive hardware/software systems*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1997.

[Ver02] A. Verdejo. Building Tools for LOTOS Symbolic Semantics in Maude. In *FORTE*, volume 2529 of *LNCS*, pages 292–307. Springer, 2002.

[vEtHSU07] M.C.J.D. van Eekelen, S. ten Hoedt, R. Schreurs, and Y.S. Usenko. Analysis of a Session-Layer Protocol in mCRL2. In S. Leue and P. Merino, editors, *FMICS*, volume 4916 of *LNCS*, pages 182–199. Springer, 2007.

[VMO02] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. *ENTCS*, 71, 2002.

[VMO06] A. Verdejo and N. Martí-Oliet. Executable Structural Operational Semantics in Maude. *J. Log. Algebr. Program.*, 67(1-2):226–293, 2006.

[WBRG08] K. Wijbrans, B. Buve, R. Rijkers, and W. Geurts. Software Engineering with Formal Methods: Experiences with the Development of a Storm Surge Barrier Control System. In J. Cuéllar, T.S.E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *LNCS*, pages 419–424. Springer, 2008.

[Web07] M. Weber. An Embeddable Virtual Machine for State Space Generation. In D. Bosnacki and S. Edelkamp, editors, *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN), Berlin, Germany*, volume 4595 of *LNCS*, pages 168–186. Springer, 2007.

[Wee07] M.J. van den Weerdenburg. An account of implementing applicative term rewriting. *ENTCS*, 174(10):139–155, 2007.

[Wei81] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE'81), San Diego, CA, USA*, pages 439–449. IEEE Computer Society Press, 1981.

[WF05] A.J. Wijs and W.J. Fokkink. From chi-t to μCLR: Combining Performance and Functional Analysis. In *ICECCS*, pages 184–193. IEEE Computer Society, 2005.

[Wie03] J. Wielemaker. An Overview of the SWI-Prolog Programming Environment. In *WLPE*, volume CW371 of *Report*, pages 1–16. Katholieke Universiteit Leuven, 2003.

[Wil11] W.W. Wilson. Implementation of Axiomatic Language. In J.P. Gallagher and M. Gelfond, editors, *ICLP (Technical Communications)*, volume 11 of *LIPIcs*, pages 290–295. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[WLBF09] J. Woodcock, P.G. Larsen, J. Bicarregui, and J.S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[Wol09] T.J.L. Wolterink. Operational Semantics Applied to Model Driven Engineering. Master's thesis, University of Twente, 2009.

Summary

The thesis presents different techniques that can be used to build formal behavioral models. If modal properties are formulated, the models can be subjected to verification techniques to determine whether a model possesses the desired properties. However many native environments do not facilitate tools or techniques to verify them. Hence, these models need to be transformed into other models that provide suitable techniques for a formal analysis. The transformations are classified into two engineering approaches, namely *syntactically engineered models* and *semantically engineered models*. Syntactically engineered models are constructed from input specifications without explicitly considering the semantics. Semantically engineered models are constructed from input specifications by explicitly considering the semantics.

The syntactic engineering approach presents four dedicated modeling techniques that construct or disseminate verification results for formal models.

The first modeling technique describes a way to create models from system descriptions that specify concurrent behavior. Here, we model three variations of a 2×2 switch, for which the models are subsequently compared to models created in the specification languages: TLA+, Bluespec, Statecharts, and ACP. The comparison validates that mCRL2 is a suitable specification language to model descriptions or specify the behavior for prototype systems.

The second syntactic technique constructs an mCRL2 model from a software implementation that operates a printer for printing Printed Circuit Boards. The model is used to advise (other) software engineers on dangerous language constructs in the control software. Hence, the model is model checked for various safety properties. The implementation is modeled through an over-approximation on the behavior by abstracting from program variables, such that only interface calls between processes and non-deterministic choices in procedures remain.

The third modeling technique describes a language transformation from the language Chi 2.0 language to the mCRL2 language. The purpose of the transformation is to facilitate model checking techniques to the discrete part of the Chi 2.0 language. The transformation illustrates that even though the languages reside in the (same)

timed discrete event domain, it is not trivial to translate all syntactic notions.

The fourth technique offers a visual solution to disseminate verification result from formal models to different disciplines using native physical designs for industrially sized systems. We demonstrate the dissemination for a practical situation, showing that these solutions add value to the validation and verification of functional behavior.

By applying these modeling techniques, we observe that all techniques require human ingenuity, which can potentially introduce unintended behavior. To reduce the chances of introducing unintended behavior and rule out the human ingenuity effort as much as possible, we propose a semantic engineering approach, that constructs formal models based on the formal semantics of a language.

We first formalize the behavior of an (informal) industrial Domain Specific Language, using a Transition System Specification (TSS). By performing the formalization in a compositional way, we show that it is possible to formalize an industrial language, and that behavioral ambiguities can be resolved when informed choices are made.

The second step describes the transformation of a TSS to a Linear Process Specification (LPS). The transformation is specified for deduction rules that are in *deSimone* format, including predicates. The LPSs are specified in the syntax of the mCRL2 language, that, with the help of the underlying (higher-order) re-writer/tool-set, can be used for simulation, exhaustive labeled transition system generation and verification of behavioral properties.

The applicability of the approach is finally demonstrated by taking on the formal definition of the (untimed) mCRL2 language. To validate that the implementation corresponds to its formal semantics we directly model the corresponding TSS. Despite its formal characterization, thorough study and broad use in many areas, the approach reveals a number of (subtle) differences between the mCRL2's intended semantics, the defined semantics and the actual implementation.

Samenvatting

Dit proefschrift beschrijft verschillende technieken die gebruikt kunnen worden om formele gedragsmodellen te construeren. Als modale eigenschappen zijn geformuleerd, kunnen de bijbehorende modellen onderworpen worden aan verificatie technieken om te zien of deze de gewenste eigenschappen bezitten. De omgevingen waarin deze modellen worden geconstrueerd, bieden vaak geen of nauwelijks faciliteiten voor verificatie. Vandaar dat deze modellen worden getransformeert naar modellen in andere omgevingen die wel geschikt zijn voor verificatie doeleinden. De beschreven transformaties worden geclacifiseerd door twee benaderingen, namelijk de traditionele *syntactische constructie van modellen* en een alternatieve *semantische constructie van modellen*. Syntactisch geconstrueerde modellen worden uit specificaties vervaardigd zonder expliciet te letten op de semantiek. Semantisch geconstrueerde modellen worden uit specificaties verkregen door expliciet gebruik te maken van de semantiek.

De informele constructie van modellen wordt beschreven door vier toegewijde technieken die resulteren in formele modellen. Deze kunnen vervolgens gebruikt worden voor validatie en verificatie doeleinden.

De eerste techniek demonstreert een case studie waarin gedragsmodellen voor een 2×2 switch worden opgesteld voor een systeem dat parallel gedrag beschrijft. De modellen worden beschreven in de mCRL2 taal, waarna deze worden vergeleken met de resulterende modellen van vier andere specificatie talen, namelijk TLA+, Bluespec, statecharts, en ACP. De vergelijking bevestigt dat de mCRL2 taal geschikt is voor het beschrijven van gedragsmodellen.

De tweede syntactische techniek construeert een mCRL2 model uit een software-implementatie die de aansturing verzorgt voor het printen van printplaten. Het resulterende model in deze case study wordt gebruikt om (andere) software engineers te informeren over potentieel gevaarlijke taalconstructies in de software controller. Het model wordt vervolgens aan de verificatie van verschillende veiligheidseigenschappen onderworpen. De modeleertechniek beschrijft een over-approximatie waarbij wordt geabstraheerd van de programma variabelen, zodanig dat alleen de communicatie tussen interfaces en non-deterministische keuzes binnen de verschillende procedures

behouden blijven.

De derde techniek transformeert taalconstructies uit de taal Chi 2.0 naar mCRL2 taalconstructies. Met behulp van de vertaling worden model-check technieken beschikbaar gemaakt aan het discrete gedeelte van de Chi 2.0 taal. Hoewel beide talen zich in eenzelfde getimede discrete event domein bevinden, illustreert de aanpak dat de transformatie niet geheel triviaal is.

De vierde techniek beschrijft een visuele oplossing om verificatie resultaten uit formele modellen te relateren aan de originele ontwerpen van industriële systemen. De aanpak wordt geïllustreerd door een case study. Uit deze oplossing komt naar voren dat soortgelijke technieken een toegevoegde waarde bieden bij de validatie en de verificatie van functioneel gedrag.

Bij uitvoering van bovenstaande modelleertechnieken observeren wij dat bij het construeren van gedragsmodellen er steeds sprake is van een menselijke ingenuïteit, een interpretatie of een inbreng, die mogelijk onbedoelde gedrag introduceert. Om de menselijke invloedsfactor te reduceren, beschrijft het tweede gedeelte van het proefschrift een structurele en semantische modelleermethode die door middel van de formele semantiek van een taal, formele gedragsmodellen construeert.

Daarvoor beschrijven we eerst hoe het gedrag van een informele taal geformaliseerd kan worden. Aan de hand van een casus wordt een industriële domein specifieke taal omgezet naar een Transitie Systeem Specificatie (TSS). Door de formalizatie compositionele uit te voeren laten we zien dat deze methode geschikt is voor een industriële taal. Daarnaast laten we zien dat een exercitie als deze ongewenst en/of onduidelijk gedrag aan het licht brengt. Door tijdens de formalizatie wel overwogen keuzes te maken kunnen deze problemen worden verholpen.

De tweede stap beschrijft de transformatie van een TSS naar een Lineaire Proces Specificatie (LPS). De transformatie wordt beschreven voor deductie regels die voldoen aan het *DeSimone* formaat, inclusief predicaten. De LPSen worden beschreven in de syntax van de mCRL2 taal, waardoor het mogelijk is om met de bijbehorende toolset en onderliggende hogere-orde herschrijver, het bijbehorende gelabelde transitie systeem te genereren of de specificatie te verifiëren.

De toepasbaarheid van de semantische methode wordt onderzocht door de formele definitie (van het tijdloze fragment) van de mCRL2 taal te nemen en deze als input te gebruiken voor deze methode. Door de operationele semantiek rechtstreeks te vertalen naar noties in de LPS, valideren wij tevens dat de beoogde semantiek en diens implementatie overeenkomen. Ondanks de formele karakterisering, de grondige studie en het brede gebruik van de taal, laat de exercitie toch een aantal (subtiele) verschillen zien tussen de bedoeld semantiek, de gedefinieerde semantiek en de geïmplementeerde executie van de mCRL2 taal.

Curriculum Vitae

Frank Stappers was born on April 24th, 1982 in Weert. In August 2000 he completed secondary school (VWO) at the Philips van Horne Scholengemeenschap in Weert. In September 2000 Frank started to study Computer Science at Eindhoven University of Technology. He obtained his Master's degree in April 2007 on his master's thesis entitled *Modeling, Validation, Verification and Integration of models with the χ-toolkit applied in an ASML case-study*.

From April 2007 Frank worked as a Ph.D. Student at the Design and Analysis of Systems Group at the Computer Science department of the Eindhoven University of Technology. The first three years he was involved in the *TWINS: Optimizing Software Hardware Co-design Flow for Software Intensive Systems* project. In the fourth year he moved to the Formal Methods Group at the Computer Science department of the Eindhoven University of Technology to carry out the "Kenniswerkingsregeling" project *LithoSysSL* at ASML. In the final year he returned to the Design and Analysis of Systems Group, which was by then renamed to Model Driven Software Engineering Group, at the Computer Science department of the Eindhoven University of Technology to finish his thesis.

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 Multi-disciplinary 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

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective*. Faculty of Mathematics and Computer Science, TU/e. 2012-12