Pattern-based refinement of models with data

van Rooij, A.R.

Award date:
2015

Master's Thesis:
Pattern-based Refinement of Models with Data

A.R. (Anson) van Rooij

Section: SET (MDSE)
Department: M&CS, TU/e

Assessment committee:
Dr. Ing. A.J. (Anton) Wijs (supervisor)
Dr. R. (Ruurd) Kuiper
Dr. E.P. (Erik) de Vink
Dr. Ir. M.A. (Michel) Reniers

Abstract

In model driven software engineering, software systems may be modeled as discrete-state systems, whose behavioural properties can be formally verified using model checking. Previous work has uncovered the need for a method to automatically refine a high-level software model to a lower-level software model which is closer to the structure of a specific implementation platform. Each transformation step should preserve the behaviour of the transformed system. Furthermore, the behaviour preservation of transformation steps should be verifiable input-independently, that is, without regard to the particular model which is transformed.

In this thesis we improve an existing pattern-based model transformation method for labelled transition systems which supports such behaviour preservation verification. We lift the existing method to the level of process algebra, which allows us to add data parameters to our model and to perform symbolic analysis. Our model specification format has been chosen with model-checking tool support in mind. We prove that our version of the method, like the existing method, supports input-independent proofs of property preservation for refinement steps. Our proofs and compositional development constructs bring the process algebra based method up to par with the latest developments in the existing method.

Acknowledgements

I would like to thank my supervisor Anton Wijs for offering me this interesting topic to work on and for his expert guidance on both the content and the process of my graduation project. I would like to thank Ruurd Kuiper, Erik de Vink, and Michel Reniers for being on my assessment committee, and Ruurd Kuiper for his small but crucial administrative role early in the project. I am also grateful to the faculty members at MDSE who advised me on various topics during this project, in particular Wieger Wesselink for his help exploring the mCRL2 toolset. The researchers and PhD candidates at MDSE, and my officemates Sanne Wouda and Jos Demmers, have made me feel right at home in the group during my project. On a broader level I thank all of my friends at Quadrivium, in Computer Science, and elsewhere for enriching my life during my studies in Eindhoven. I am very grateful to my parents for all of their patience and support.

Contents

Chapter 1

Introduction

1.1 Relevance

Model Driven Software Engineering (MDSE) is the practice of using system models or domain models in the software engineering process. MDSE is a varied and dynamic field: a model may be close to the user or close to the hardware, and it may model entities and relations or states and transitions (compare the *Unified Modeling Language (UML)* to *Finite State Machines (FSM)* for an example of two very different modeling languages). Goals of MDSE include easing communication about a design, standardizing and reusing design patterns, (semi-)automatically generating software, and extracting and analysing the properties of an existing system.

Many research programs include research on MDSE. The *ECSEL* public–private partnership is a European Union program for research in electronic components and systems as a key enabling technology for all industrial branches and many aspects of life, including smart mobility, energy and health [3].

A part of ECSEL is the *ARTEMIS* embedded computing initiative. ARTEMIS sponsors pilot programmes, including the programme *AIPP5* on the topic of computing platforms for embedded systems [1], which includes the sub-project EMC^2: Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments. EMC^2 aims to find solutions for the problems of adaptability, scalability, and life cycle management in mixed (parallel) real-time systems [2].

This thesis summarizes an MDSE research project which is part of an EMC^2 line of research in which a model transformation method is developed for time- and cost-efficient evolution and adaptation of discrete system models in order to generate high-quality software systems in a mathematically rigorous way.

1.2 Motivation

The current line of research started with a project which sought to generate software for embedded platforms from models in the state machine modeling language SLCO, in the PhD-thesis of Engelen [13].

It soon became apparent that the various target platforms had computation primitives and communication interfaces which differed so much that a strategy was required to dependably adapt a given system for execution on different platforms without risky and time-consuming manual work. Based on these requirements, and inspired by work on graph transformation, an *input-independent verification method for model-to-model transformations* has been developed by Engelen and Wijs [14].

The developed method operates on the *networks of LTS* formalism [20].

The specific aim of this new model transformation method was to provide a model-based formalism for *refining* a high-level system model which results from a straightforward reading

of the system requirements to various different low-level system models which are suitable for code generation or other applications. The model transformation (/refinement) method has been implemented in the toolset REFINER by Wijs [31].

The formal correctness of the LTS-based model transformation method has since been verified, and the method made more reliable, by de Putter [25]. In recent work, the method has been expanded to make it more general (for example, by enabling the addition of new parallel components) and reduce proof obligations for transformations (for example, by allowing reasoning over subsystems) by Wijs [29].

Important limitations of the existing method are that it cannot model data as such, it cannot model any infinite-size behaviour resulting from the inclusion of data (such as the passing of arbitrary data values) and it lacks symbolic modeling options generally. We aim to add to the existing work by lifting the existing method to a symbolic level, so as to make it more general (both in modeling and transforming) and giving the proof system a link to other existing formal methods work which can be exploited to make verification more powerful.

1.3 Related work

1.3.1 Classification

Plenty of survey papers classify model transformation (verification) methods, but many of them treat models only from the perspective of the *metamodeling* paradigm. Still their categorizations may be helpful, so we give three examples.

Czarnecki & Helsen [9] [10] present a metamodeling-based classification of model transformation methods. Based on the definitions by Czarnecki & Helsen [9] we classify our transforation method as follows:

Rule format is pattern-based, syntactically separated Left Hand Side/Right Hand Side system, and this thesis introduces a term format for patterns (where the existing method uses graphs);

Scoping happens on the level of processes;

Updates are in-place and possibly destructive on the existing model;

Application is on one point and nondeterministic;

Scheduling is iterative; the rule system is assumed to be confluent;

Organization of the rules follows the same network structure as that of source and target systems; patterns can be moved between rule systems.

Traceability is not currently part of the method;

Bidirectionality is introduced into the method in this thesis.

Amrani et al. [4] present a metamodeling-based survey and classification of model transformation verification methods based on three dimensions: transformations, property kinds, and verification techniques. We covered the classification of transformations already. Based on the definitions by Amrani et al. We classify our verification method as follows in the other two categories:

Property kinds: Our collections of transformation rules are terminating and deterministic, because they are required to be confluent. Our transformations are 1-to-1 and we verify that they are semantically preserving (which we use to guarantee preservation of temporal properties);

Verification techniques: Our verification is transformation-dependent and input-independent, that is, we verify not a format, but each specific transformation, and such a transformation can then be applied to any input model. We use model-checking to perform the verification.

Rahim & Whittle present a survey and classification which does take multiple transformation definition paradigms into account. It distinguishes testing, theorem proving, graph theory and model checking. Our method combines elements of graph theory and model checking, and can be classified in the schema of Rahim & Whittle as *directly* verified (we verify the rule, not the output model) and formal.

Rahim & Whittle [26] distinguish various notions of correctness: type correctness, preservation of static semantics, preservation of dynamic semantics, correspondence between source and target, and semantics of model transformations.

The latter category corresponds roughly to Amrani et al.'s concept of transformation-independent verification. We prove some properties in the category semantics of model transformations in this work. Our transformation method itself can be classified as preserving the dynamic semantics of a model.

Below we review some of the surveyed literature in the categories of graph theory and direct model checking.

1.3.2 Graph theory

The existing refinement method is inspired by work on *graph transformation*. Graph transformation usually consist of patterns and are applied using the *double-pushout* approach. Source and target graph, and left and right pattern are mapped to each other through a subset of their vertices. This is echoed in the concept of *glue variables* in this thesis.

Graph-based approaches have the disadvantage that graphs which are used as models do not have a natural semantics. Therefore graph transformation verification is limited to for example correspondence between source and target (e.g. [11] [27]) or high-abstraction-level reasoning about the semantics of model transformations (e.g. [19], [18]).

In graph transformation it is customary to apply a transformation to a graph iteratively. This means that a model transformation may create new application options for itself in the output model. On the other hand, in general MDSE a more common matching approach is *match once–replace once*: a set of transformation matches is found once, and then all transformations are performed simultaneously (with some form of conflict resolution). Newly created application options are not searched for, so transformation always terminates. The latter is our approach.

1.3.3 Model checking

Finding properties and semantic equivalences in algebraic models is a form of *model checking*. Model checking is an approach to software verification and as such it is an alternative to testing. Where testing involves trying out a limited number of inputs and interactions in order to investigate whether a system adheres to its requirements, model checking uses a formal, mathematical approach to verification which ensures that we can reason about its properties with mathematical precision.

Like those on graph transformation, many works on direct model checking only concern themselves with correspondence between source and target (e.g. [21]) or with the semantics of model checking (e.g. [28], [32]), sometimes in combination with preservation of static semantics (e.g. [15]). One example of model checking-based transformation rule verification which is concerned with preservation of dynamic semantics is found in work by Boronat et al. [6] where transformation sets are represented as theories in rewriting logic, and rewriting paths are analysed for preservation of individual properties. Our work relies on confluence of rule systems but is conceptually simpler and checks preservation of the complete behaviour of a system at once.

Model checking is an important part of software verification for safety- and security-critical systems, because it provides an alternative and very precise view of a system's behaviour. But once software systems, particular ones involving multiple parallel components, become large, model checking suffers from the so-called *state-space explosion problem*: the complexity of the system's behaviour grows exponentially with the number of variables and components involved, and checking all of its states becomes infeasible.

1.3.4 Advanced model checking

Symbolic model checking abstracts from the state space of a model, investigating the model at the level of its description. An example of symbolic model checking which is relevant to this thesis is work by Chen et al. [7] on the reduction of equivalence test for algebraic specifications to systems of equations called parameterized boolean equation systems which can subsequently be solved by tool or by hand.

Symbolic model checking mitigates the state-space explosion problem somewhat, but cannot remove it completely. When developing software, model checking at each iteration, whether symbolic or not, remains a time- and cost-inefficient task for most applications.

By verifying the correctness of transformation rules we introduce two further reductions in the effort required to check the correctness of a series of iteratively developed models using model checking:

- The specification of a transformation rule is necessarily smaller (usually much smaller) than that of the complete model under transformations.
- Once verified, an appropriately defined transformation rule can be reused arbitrarily often in different models without further verification effort.

There is another approach to creating, rather than verifying, models using formal techniques from the model checking paradigm, and it has similar advantages: automatic generation of (part of) a model. Topics of existing work include the generation of synchronization skeletons [12], automated functional refinement guided by counterexamples [8], and updating a model to satisfy new properties [33].

A clear difference with our method is that we are concerned specifically with preserving behavioural properties of a system while changing structural properties (like distributing functions over a set of parallel processes in a certain way), while model generation methods are designed to change behavioural properties while not caring about structural properties. Such approaches could be complementary to our method in generating a model from which to start transformation-based development.

1.4 Project goals

Our challenge in this project is to find a formalism in which we can model discrete (software) systems and their behaviour, can check their properties, and can define model transformations. The formalism should be symbolic, it should be able to model data, and it should support a refinement method which has all of the power of the existing LTS-based method. We should be able to show how definitions and transformations in the existing method can be mapped to the new method.

Our approach to this challenge is to use *process algebra* as our formalism for modeling systems. Process algebra is a mathematical formalism intended for reasoning about parallel processes. We use replacement of equivalent sub-expressions as our model transformation method (where the equivalence of the system of a whole must be maintained during replacement of equivalent parts).

Process algebra is a natural choice of approach for our purposes, being in its most basic form a symbolic way of modeling LTS, which means that finding equivalents of the definitions and transformations of the existing LTS based method is a natural affair. Process algebraic tools for modeling data already exist.

We show that process algebra can indeed be used to define models and transformations in ways equivalent to those of the existing method. We show that symbolism, data, and some other extensions can also be defined in a natural way. Some of our proofs of core requirements for model transformation are arguably simpler than those of the existing method. We further show the applicability of our method using several examples. We make it feasible to transform the resulting models with more complex and realistic features, broadening the theoretical basis of the model refinement method.

Thesis overview In Chapter 2 we review preliminary concepts. In Chapter 3 we show how process algebra can be used to correctly refine single processes. In Chapter 4 we show how process algebra can be used to correctly refine networks of LTS. In Chapter 5 we extend our lifted method to handle data, irregularity, and more complex refinement steps. In Chapter 6 we present our conclusions and suggest directions for future work.

Chapter 2

Preliminaries

2.1 Labelled Transition Systems (LTS)

Earlier work operates on the Labelled Transition System (LTS) formalism. An LTS is an edge-labelled directed graph with an initial state. LTSs are commonly used to represent the state space of a software system.

Definition 1 (Labelled Transition System)**.** *A Labelled Transition System G is a tuple (S_G, A_G, T_G, I_G) with*

S_G *a finite set of states;*

A_G *a finite set of actions;*

$T_G \subseteq S_G \times A_G \times S_G$ *a finite set of transitions;*

$I_G \subseteq S_G$ *an initial state.*

We are interested in constructing systems which exhibit certain properties when interacting with the rest of the world. Important such properties are for example safety properties ('Bad things will not happen.') and liveness properties ('Good things will eventually happen.'). A usually undesirable state of systems is deadlock ('No further action is possible.'), and so a commonly required liveness property is non-deadlock ('The system will never reach a deadlock state.').

We are interested in both the interaction properties of systems and their structural properties, for example, the division of various functional parts of the system over different processes. Different systems with exactly the same interaction properties but possibly different structural properties are called bisimilar. We check whether two systems are bisimilar by trying to find a so-called bisimulation relation between them.

There are various kinds of bisimulation relations, distinguished by the precision with which they equate structurally different systems. For example, some systems may contain transitions which are internal and can not be observed directly by the outside world. We are interested in bisimulation relations in which such internal actions, commonly referred to by the name τ, are only taken into account when they are observable indirectly because they can influence the possibility of future observable actions (and therefore of future interaction).

These kinds of bisimulation relations are called branching bisimulation relations, as defined in Definition 2. We abbreviate 'S is branching bisimilar to R' by the expression $S \underline{\leftrightarrow}_b R$.

Definition 2 (Branching bisimulation)**.** *A binary relation B between two LTSs G_1 and G_2 is a branching bisimulation iff for all $s \in S_{G_1}$, $t \in S_{G_2}$, sBt implies that s can* simulate *all behaviour of t, and vice versa:*

1. *If $(s, a, s') \in T_{G_1}$ then*

- *either* $a = \tau$ *and* $s'Bt$*;*
- *or there exists a finite sequence of states* $t_l \in S_{G_2}$ *with* $l \in [0..n]$ *such that* $t = t_0$*,* $(t_l, \tau, t_{l+1}) \in T_{G_2}$ *for all* $l < n$*, and* $(t_n, a, t') \in T_{G_2}$*, with* sBt_n *and* $s'Bt'$*;*

2. *and vice versa.*

The μ-calculus is a formal language for expressing properties of systems, expressive enough to model safety and liveness properties. Mateescu and Wijs identified a fragment of the μ-calculus which is adequate with branching bisimulation [23]. This fragment can be used to specify properties of the systems to be refined.

We do not define the μ-calculus in this thesis, because the exact properties which are to be preserved by a transformation play a relatively minor role in this project. Property preservation is determined (after hiding actions irrelevant to the property [23]) by checking bisimulation preservation, which is not concerned with specific properties.

2.2 Process algebra

We need a way to model systems composed of LTSs, but possibly containing data. Process algebra is a family of languages and calculi which can be used to reason about concurrent systems, and on the most basic level, the model of process algebra is the LTS. In this project we model LTSs using (subsets of) the process algebra language mCRL2, which contains constructs for modeling the data state (in addition to the process state) of a system [16].

One of the benefits of using process algebra in modeling is that we have straightforward ways to establish useful properties like congruence.

Definition 3 (Congruence)**.** *An equivalence relation* R *between algebraic expressions is a congruence if it compatible with the structure, that is, if for* $\forall e_1, e_2$ *with* $e_1\ R\ e_2$*,* $\forall s_1, s_2$ *with* s_1 *a subexpression of* e_1 *and* $s_1\ R\ s_2$*, we have* $e_1[s_1 \setminus s_2]\ R\ e_2$*.*

It turns out that in the presence of internal actions branching bisimulation is a congruence on process algebras like mCRL2 if it satisfies the rootedness condition given in Definition 4 [5].

Definition 4 (Rootedness)**.** *A branching bisimulation relation* B *between two LTSs* G_1 *and* G_2 *with initial states* s *and* t*, respectively, satisfies the rootedness condition iff for all* $s' \in S_{G_1}$*,* $t' \in S_{G_2}$*,* sBt *implies:*

1. *If* $(s, a, s') \in T_{G_1}$*, then there exists a* $t' \in S_{G_2}$ *such that* $(t, a, t') \in T_{G_2}$ *and* $s'Bt'$*;*
2. *and vice versa.*

We refer to a branching bisimulation relation which satisfies the rootedness condition as a *rooted branching bisimulation*, abbreviated $\underline{\leftrightarrow}_{rb}$.

Chapter 3

Single process transformations

In this chapter we establish a method of refining systems which consist of mCRL2 processes without data or parallelism. In general refining a system means transforming the system to a more structurally complex form without changing its observable properties.

We interpret maintenance of observable properties formally as preservation of (rooted) branching bisimulation. We prove that our transformations preserve (rooted) branching bisimulation if they conform to a certain format and pass a certain correctness check.

A system becomes more structurally complex if it gains components or communications between components. In this chapter we do not yet have parallelism, so transformations will in fact not yet increase the complexity of systems in a significant way.

We prove that our transformations can transform any single-process system into any other process in its rooted branching bisimilarity class.

3.1 BPA process definitions

In this chapter we specify processes using the part of process algebra mCRL2 without data or parallelism, which is called Basic Process Algebra (BPA). The signature of BPA contains a set of actions and a set of process variables, as described in chapter 2. The two binary operators in the signature of BPA are given in Table 3.1.

Symbol	Meaning
$+$	choice (alternative composition)
$\cdot$	sequential composition

Table 3.1: The operators of BPA

An algebra also has a set of axioms. A set of axioms only holds in the context of (and in fact defines) a certain equivalence. The axioms of BPA which correspond to (rooted) branching bisimulation are given in Table 3.2.

In addition to requiring that the expressions which define our processes conform to the signature of our algebra BPA, we also require that processes are specified in a format which we call the pCRL format, as defined in Definition 5. This format (a simplified version of the existing pCRL format [17]) makes communication about processes easier, and the use of pCRL specifically is of great practical importance because it translates readily into the mCRL2 specification format, which enables us to use the mCRL2 toolset for model checking.

Definition 5 (pCRL format)**.** *A BPA process specification S in pCRL format is a tuple $\langle A_S, V_S, P_S, I_S \rangle$ with*

Label	Axiom
A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y) \cdot z = x \cdot z + y \cdot z$
A5	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$
A6	$x + \delta = x$
A7	$\delta \cdot x = \delta$
W	$x \cdot \tau = x$
BRANCH	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$

Table 3.2: The axioms of BPA

Action set A_S. *The unobservable, internal action* τ *and the deadlock* δ *are always part of the action set, and do not need to be specified;*

Process variable set V_S. V_S *contains every process variable defined in* P_S*;*

Process definition set P_S. *A set of definitions which can be viewed as a function from elements of* V_S *to definitions in the form of process expressions;*

Initialization expression I_S. *A single process variable: the initial variable.*

For an example of a process specification, see the example in Section 3.5. We define some restrictions on the input format for our transformations.

The pCRL format given in Definition 5 does not allow only *closed* process specifications, as defined in Definition 6. In other words, the pCRL format also allows *open* specifications. If a process specification is not closed, its possibilities to perform actions is not fully defined and it cannot be assigned a (rooted) branching bisimilarity class. We assume that every process specification which we transform is closed.

Definition 6 (Closed)**.** *A subset* $V'_S \subseteq V_S$ *is closed iff some subset* $P'_S \subset P_S$ *is total on* V'_S *and no* $V_i \in V'_S$ *occurs in a definition* $P_j \notin P'_S$*. A process specification* S *is closed if* $V'_S = V_S$ *is closed with* $P'_S = P_S$*. Otherwise* S *is open.*

In a closed process specification, V_S can be inferred from P_S. Therefore we omit V_S from examples of process specifications.

The LTS which is modeled by a process specification is called its *solution*. To make sure that a process specification falls into one (rooted) branching bisimilarity equivalence class we want to make sure that it has exactly one solution. Process specifications without recursion always have exactly one solution, but the same does not hold for process specifications with recursion in general.

It has been established however that a process specification with recursion does have exactly one solution if it is closed and regular [5] as defined in Definitions 6 and 8. Lemma 1 says that there exists a map which associates every closed, regular process specification with its solution. The other way around, the same map associates every LTS with a closed, regular process specification.

Lemma 1. *There exists a bidirectional map between LTSs to equivalent process specifications.*

Proof. We define a bidirectional mapping *Reg*. *Reg* associates each LTS $G = \langle S_G, A_G, T_G, I_G \rangle$ with an equivalent process specification S with $\langle A_S, V_S, P_S, I_S \rangle$, where

- $A_S Reg A_G$;
- $V_S Reg S_G$;

- P_S defines each $S \in V_S$ with at least one summand: δ. Furthermore, P_S associates with each transition (S, a, T) in T_G a summand $a \cdot Reg(T)$ in the definition of S;
- $I_S = I_G$.

□

Now that we have established that every LTS can be transformed to a closed, regular BPA process specification, we know that the class of closed, regular process specifications is expressive enough for our purposes: it can model any LTS. While working in BPA we assume that every process specification which we transform is guarded (as defined in Definition 7) and regular. We call two such process specifications (rooted) branching bisimilar if their solutions are (rooted) branching bisimilar.

Definition 7 (Guarded). *An occurrence of a process variable P in a process expression is guarded if P is sequentially composed after an action. A process specification S is guarded if all occurrences of process variables in defining process expressions in S are guarded.*

Definition 8 (Regular). *A process expression is regular if it is an alternative composition in which the summands (alternatives) are guarded process variables, as defined in Definition 7. In other words, every expression in a regular specification is of the form '$a_0 \cdot P_0 + a_1 \cdot P_1 + \ldots + a_n \cdot P_n$', with a_i in A_S and P_i in P_S for all $i \leq n$. A specification S is regular if every definition in P_S is regular.*

3.2 Single process transformation rules

We transform process specifications by applying a *transformation rule*, as defined in Definition 9. A transformation rule consists of a left pattern and a right pattern, which are required to be (rooted) branching bisimilar. We replace the left pattern by the right pattern in the original process specification, which is thereby transformed into a (rooted) branching bisimilar resulting process specification. Transformation will be defined more precisely in this section.

Definition 9 (Transformation rule). *A transformation rule T is of the form $\langle L, R \rangle$, with L_T its left pattern and R_T its right pattern. Patterns L and R are regular process specifications which have the same action set.*

Variables in the intersection between V_{L_T} and V_{R_T} are called the glue variables *of T.*

For an example of a transformation rule, see the example in Section 3.5. To apply a transformation rule to a process specification, we first need to find a *match* of the left pattern on the original system, as defined in Definition 10. Informally, a match can be understood as a subprocess isomorphism.

Definition 10 (Match). *A match m between pattern L of some transformation rule $T = \langle L, R \rangle$ and some process specification S is an injective map from pattern variables to a closed subset of process variables of S which has the following property:*

- *For all pattern variables $V \in V(L)$ with definition $P_L(V)$ and match $m(V) = V'$, replacing all pattern variables $V_i \in P_L(V)$ by their match $m(V_i)$ yields a new definition $P_S(V')$ equal to $P_L(V)$ modulo associativity and commutativity of operators.*

We formally define the effect of an *application* of a transformation rule to a process specification in Definition 11. For an example of a transformation rule application, see the example in Section 3.5.

Definition 11 (Application)**.** *In an application of transformation rule T to a process specification S, the left pattern L_T must have a match m on S. The resulting process specification is called $T(S,m)$, on which the right pattern R_T has a match m'.*

For each glue variable V of T we require that $m(V) = m'(V)$, so all matches of glue variables remain in the process after the application. To avoid variable collisions we assume that process variable set V_S does not overlap with pattern variable sets V_{L_T} and V_{R_T}.

The resulting process specification $T(S,m)$ of a transformation is as follows:

$A_{T(S,m)}$ $= A_S \cup A_{R_T}$*;*

$V_{T(S,m)}$ $= V_S \setminus m(V_{L_T}) \cup m'(V_{R_T})$.

$P_{T(S,m)}$ *is total on* $V_{T(S,m)}$ *and maps each* $V \in V_{T(S,m)}$ *to:*

- $P_{R_T}(m'^{-1}(V))$ *if* V *is a target of* m'*;*
- $P_S(V)$ *otherwise;*

$I_{T(S,m)}$ $= I_S$.

3.3 Connections between match and context

The patterns of a transformation rule match only part of a process. The original and resulting processes of a transformation may enter or leave the matches of a transformation's patterns at the transformation's glue variables. To check whether a transformation preserves (rooted) branching bisimilarity, we need to represent the expressions at the edges of the transformation's matches, which make a match open (not closed). To that end we introduce the concept of *entry* and *exit* variables, as given in Definition 12. A process variable may be both an entry variable and an exit variable.

Definition 12 (Entry/Exit variables)**.** *The entry variables and exit variables are subsets of the glue variable set of a rule. Entry and exit variables are exceptions to the property that match m of pattern T on process specification S is a closed subset of V_S, in the following way:*

- *For process variables $V_i, V_j \in V_S$ with $V_i \in m(T)$ and $V_j \notin m(T)$, if and only if V_i is an entry variable may V_i occur in a summand sum_i of $P_S(V_j)$. The initial variable of a process definition must be matched by an entry variable, if it is matched at all.*
- *For process variables $V_i, V_j \in V_S$ with $V_i \in m(T)$ and $V_j \notin m(T)$, if and only if V_j is an exit variable may V_i still occur in a summand sum_i of $P_S(V_j)$. Summands not occurring in the pattern are preserved by transformation rule application.*

To show which variables are entry and exit variables, we introduce the special process variable V_κ. We extend pattern specifications with V_κ and with κ-actions. Entering or leaving a match is modeled by κ-actions. κ is a reserved action, and is added to the process specification in the way defined in Definition 13.

Definition 13 (V_κ variable and κ-extension)**.** *A κ-extended pattern P^κ contains process variable V_κ. V_κ replaces the original initial variable of the pattern as the initial variable of the extended pattern, and its definition is regular. A κ-extended pattern also contains κ-actions, each labeled with a unique integer i. κ-actions may occur in the following ways:*

- *For each entry variable V_i, including the original initial variable, there exists a summand of $P_P(V_\kappa)$ which is equal to $\kappa_i \cdot V_i$.;*
- *Each exit variable V_o has a summand $\kappa_i \cdot Synch$ in $P_P(V_o)$.*

Besides ensuring that it is clear which variables can connect to the unmatched part of the system, the uniqueness of κ actions ensures that different entry and exit variables are identifiable individually, even though these may not have distinguishing definitions in the pattern specification.

κ-extension resolves another issue. Recall that the rootedness condition of Definition 4 was introduced to deal with cases in which part of an LTS transition is replaced by a branching bisimilar series of transitions starting with at least one τ-transition (which is allowed by congruence), while other actions remain unchanged. The resulting LTS would not be bisimilar to the original LTS, because taking the new τ-transitions would disable the unchanged behaviour. Lemma 2 says that we do not need to check the rootedness condition in certain cases.

Lemma 2. *In any application of a κ-extended transformation rule where the extended patterns are branching bisimilar, the rootedness condition is also preserved.*

Proof. The rootedness condition was introduced because part of the transitions of a state may be replaced by τ-transitions, creating an invisible step which may make other transitions unreachable. In regular process specifications the transitions of an LTS state are represented by the summand set of a process variable.

Exit variables are the only process variables where part of the summand set may be changed while another part remains the same. We show that in exit variables no τ-prefixed summands can be introduced which make other summands unreachable.

All exit variables have κ-prefixed summands which must be preserved, which means that a bisimulation relation on the patterns will have to preserve the possibility of taking these κ-actions.

If replacing a summand sum by a τ-prefixed summand sum_τ introduces an invisible choice which makes certain other summands unreachable, then sum_τ causes this unreachableness for all summands with an action different from that of sum.

In the patterns as defined in this thesis no κ-actions are replaced, so no replaced sum will contain any κ-action, so introducing a τ-prefixed summand will always disable any κ-prefixed summands, so no τ-prefixed summands can be introduced in exit variables.

Since non-exit variables will not have part of their summands replaced while other summands remain the same, unrootedness is never an issue. □

3.4 Preservation of branching bisimulation

Our goal in this chapter was to show that we can define transformation rules of which we can verify that they preserve rooted branching bisimilarity during any application on a valid match. We can now give this proof.

To check that the application of a transformation rule preserves rooted branching bisimilarity, we need to check that the left pattern is rooted branching bisimilarity to the right pattern. We prove Proposition 1 which makes this statement more precise.

Proposition 1. *For transformation rule T with ${L_T}^\kappa \underline{\leftrightarrow}_{rb} {R_T}^\kappa$, an application of T on process specification S at match m of L_T results in a process specification $T(S,m)$, where $S \underline{\leftrightarrow}_{rb} T(S,m)$ and R_T matches $T(S,m)$ at m'.*

Proof. Process specifications S and $T(S,m)$ are regular, so every transition of state s to state t in their solutions is represented by a summand in the process specifications. We use the term *behaviour* to refer to the ability to perform an action a, as per the conditions of branching bisimilarity in Definition 2. To show that $S \underline{\leftrightarrow}_{rb} T(S,m)$ we must show that $I_S \underline{\leftrightarrow}_{rb} I_{T(S,m)}$ and all behaviour of transitions of solutions of S and $T(S,m)$ is preserved modulo branching.

We build a relation C which will be our proposed branching bisimulation relation. First we define a relation A between all unmatched variables and themselves. Next, recall that we have established a branching bisimulation relation between ${L_T}^\kappa$ and ${R_T}^\kappa$, which we call the pattern relation $\hat{B}$. We define a relation B between $m(V)$ and $m(V')$ for all V, V' in $\hat{B}$. Now we define C as $A \cup B$. In the remainder of this proof we establish that C is indeed a rooted branching bisimulation relation.

We show that $I_S \leftrightarrows_{rb} I_{T(S,m)}$: $I_{T(S,m)} = I_S$ holds by Definition 11 of application, so if I_S is not matched then $I_S \in A$, and if I_S is matched then it is a glue variable, and $I_S \in B$. Lemma 2 ensures that rootedness is not an issue.

Next we show that all behaviour of transitions of S's and $T(S,m)$'s solutions is preserved. To start we establish some properties of the matches. For every variable $V \in V_S$ in the target of match m there exists a source variable $m^{-1}(V) \in m(L_T)$, and for every variable $V' \in m_{T(S,m)}$ in the target of match m' there exists a source variable $m'^{-1}(V') \in m(R_T)$.

Consider first an arbitrary target variable V of match m.

- If V has summand $sum = a \cdot W$ which exits the target of match m, then match source $m^{-1}(V)$ is an exit variable and therefore a glue variable. So there exists a variable V' in the target of match m' with source $m'^{-1}(V') = m^{-1}(V)$. The summand sum (which is represented by a κ-action prefixed summand) is preserved exactly by the application of the transformation rule. So V' has summand $sum' = a \cdot W$. So the behaviour of summands sum which exits the pattern is preserved by C.

- If V has summand $sum = a \cdot W$ which does not exit the target of match m, then sum must be present in match source $m^{-1}(V)$. Source $m^{-1}(V) \hat{B} m'^{-1}(V')$ for some V'. We distinguish two cases:

  - $a = \tau$: Then we have $m^{-1}(W)\ \hat{B}\ m'^{-1}(V')$, so $W\ C\ V'$ and the behaviour of sum is trivially preserved by the existence of V;
  - V' starts a series of τ summands $V_0 \ldots V_n$ in the pattern in which for all $i < n$, V_i has summand $\tau \cdot V_{i+1}$ and for variable V_n it holds that $m^{-1}(V_n)\ \hat{B}\ m'^{-1}(V)$, and V_n has summand $a \cdot W'$ with $m^{-1}(W')\ \hat{B}\ m'^{-1}(W)$. So the behaviour of sum is preserved by the series of summands ending in $a \cdot W'$.

  In either case the behaviour of summand sum which does not exit the pattern is preserved by C.

We have established that behaviour of any arbitrary target V of match m is preserved by C. Now consider an arbitrary variable $V \in V_S$, V not a target of match m.

- If V has summands $sum = a \cdot W$ which enters the match target of m, then V is preserved by the transformation. As for W: match source $m^{-1}(W)$ is an entry variable and therefore a glue variable. So variable W is also in the target of right match m' with source $m'^{-1}(W) = m^{-1}(W)$ (as guaranteed by a κ-action prefixed summand). So V has summand $sum' = a \cdot W$. So the behaviour of summand sum which enters the pattern is preserved by C.

- If V has summand sum which does not enter the target of match m, then sum is unaffected by the application of the transformation rule. So the behaviour of summand sum which does not enter the pattern is preserved by C.

We have now also established that behaviour of any arbitrary variable outside the target of match m is preserved by by C. So all behaviour of the original process specification S is preserved by C. Since the definition of a match is symmetric, we can apply the same reasoning to the existence of all behaviour of resulting process specification $T(S,m)$ in S. So C is indeed a rooted branching bisimulation relation. □

Remark 1 (Contributions). The specification format used in this chapter is taken from the literature. We avoided including the exact semantics of the specification formats in this thesis, in order not to make this thesis unnecessarily long.

The concepts introduced for the correctness check include the variable V_κ. Using a process variable for context-connections of single processes is new relative to the existing method and streamlines some later exposition.

We lifted the correctness check from the LTS level to the process algebra level. We show that rooted branching bisimilarity is preserved by application of transformation rules whose patterns are simply branching bisimilar. Rooted branching bisimilarity has not been treated before in work on the existing method.

The method defined in this chapter is expressive enough to transform any process specification S into any other process specification $T(S, m)$ which is bisimilar to S. Simply define a transformation rule $\langle S, T(S, m)\rangle$.

3.5 Example

Example 1. We define a transformation $T1 = \langle L1, R1\rangle$ as follows:

$L1:\ A ::= \{a\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1$

$V_1 = a \cdot V_2;$

$V_2 = \kappa_2 \cdot V_\kappa$

$I ::= V_\kappa$

$R1:\ A ::= \{a\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1$

$V_1 = \tau \cdot V_3;$

$V_3 = a \cdot V_2;$

$V_2 = \kappa_2 \cdot V_\kappa$

$I ::= V_\kappa$

Note that in transformation rule $T1$, process variable V_κ is the initial variable, and process variable V_2 is the only exit variable. The initial variable and exit variable are shared by BPA specifications $L1$ and $R1$, and are marked by κ actions.

Given these specifications of the patterns of transformation rule $T1$ we can check whether the patterns are bisimilar. After finding out that the patterns are indeed bisimilar, we can apply transformation rule $T1$ to arbitrary system specifications. For example, consider the following system specification:

$S1:\ A ::= \{a\}$

$P ::=$

$P_1 = a \cdot P_2;$

$P_2 = b \cdot P_1 + c$

$I ::= R_1$

We apply transformation rule $T1$ to system $S1$ with match $\{(V_1, P_1), (V_2, P_2)\}$ to get the following transformed system $S1' = R1(S1)$:

$S1':\ A ::= \{a\}$

$P ::=$

$P_1 = \tau \cdot V_3;$

$V_3 = a \cdot P_2;$

$P_2 = b \cdot P_1 + c$

$I ::= R_1$

The bisimilarity of the patterns of transformation rule $T1$ ensures that the original system $S1$ and the resulting system $S2$ are also bisimilar.

Chapter 4

Process network transformations

In this chapter we establish a method of refining systems which consist of multiple parallel processes. The method has the same structure as the one defined in Chapter 3: we define transformation rule systems and verify bisimulation between the patterns of these rules.

The rule systems with which we transform process networks can introduce internal communication options in the form of synchronization laws. This means that we can actually speak of refinement in the case of network transformation.

With the definitions give in this chapter we can perform network transformations which perform the same operations as those in the original LTS-based method, but this time on the level of process algebra.

4.1 The algebra ACP

In this chapter we specify processes using the part of process algebra mCRL2 which contains expressions with parallelism, which is called the Algebra of Communicating Processes (ACP).The signature of ACP contains BPA, as described in chapter 3. New binary operators in the signature of ACP are given in Table 4.1.

Symbol	Meaning
$\|\|$	merge (parallel composition)
$\lfloor\!\lfloor$	left merge
$\|$	synchronization

Table 4.1: The operators of ACP

All operators are binary.

Merge operator $||$ represents the parallel execution of two processes. For example, $P||Q$ models the parallel execution of processes P and Q.

Left merge operator $\lfloor\!\lfloor$ represents the parallel execution of two processes, where first the left side takes a single step, and then both processes proceed in parallel. The left merging operator is generally not used in specifications, but exists in order to enable us to more easily distribute the merge operator over its subexpressions.

ACP models not just interleaving concurrency, where multiple systems can take action in turn, but also true concurrency, where multiple processes take actions at the exact same time. When multiple of the individual processes can take actions at the same time, then the parallel composition of these processes can perform the combination of these actions.

This phenomenon of combined actions is called a *multi-action*, and is noted using the synchronization operator $|$. For example, $a|b|c$ is a multi-action in which actions a, b and c are performed

at the same time. The synchronization operator is generally only used in the process expressions, but results from the distribution of the merge operator over its subexpressions.

The additional axioms of ACP which describe the behaviour of the new operators are given in Table 4.2, for α and β arbitrary multi-actions. In addition to requiring that the expressions which

Label	Axiom
M	$x \| y = x \lfloor\!\lfloor y + y \lfloor\!\lfloor x + x \mid y$
LM1	$\alpha \lfloor\!\lfloor x = \alpha \cdot x$
LM2	$\delta \lfloor\!\lfloor x = \delta$
LM3	$\alpha x \lfloor\!\lfloor y = \alpha \cdot (x \lfloor\!\lfloor y)$
LM4	$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z = y \lfloor\!\lfloor z$
S1	$x \mid y = y \mid x$
S2	$(x \mid y) \mid z = x \mid (y \mid z)$
S3	$x \mid \tau = x$
S4	$\alpha \mid \delta = \delta$
S5	$(\alpha \cdot x) \mid \beta = \alpha \mid \beta x$
S6	$(\alpha \cdot x) \mid (\beta \cdot y) = \alpha \mid \beta (x \cdot y)$
S7	$(x + y) \mid z = x \mid z + y \mid z$
TC1	$(x \lfloor\!\lfloor y) \lfloor\!\lfloor z = x \lfloor\!\lfloor (y \| z)$
TC2	$x \lfloor\!\lfloor \delta = x \cdot \delta$
TC3	$(x \mid y) \lfloor\!\lfloor z = x \mid (y \lfloor\!\lfloor z)$

Table 4.2: The additional axioms of ACP

define sets of parallel processes conform to the signature of our algebra ACP, we also require that sets of parallel processes are specified in the parallel pCRL format, which is an extension of the pCRL format, as defined in Definition 5.

Definition 14 (Parallel pCRL format)**.** *An ACP specification S in parallel pCRL format is a tuple $\langle A_S, V_S, P_S, I_S \rangle$ with*

Action set A_S. *The unobservable, internal action τ and the deadlock δ are always part of the action set, and do not need to be specified;*

Process variable set V_S. *V_S contains every process variable defined in P_S;*

Process definition set P_S. *A set of definitions which can be viewed as a function from elements of V_S to definitions in the form of process expressions. Only BPA operators occur in the definitions;*

Initialization expression I_S, *which consists of a process expression.*

Note that like the pCRL format, the parallel pCRL format allows irregular processes. When working with ACP we still assume that transformed specifications and transformation rule pattern are regular. We also still assume that specifications are closed.

4.2 Alphabet operators

ACP also contains the alphabet operators. A number of alphabet operators are given in Table 4.3.

All alphabet operators are unary, but they do each have a parameter. Definition 15 briefly explains each alphabet operator. The behaviour of the alphabet operators is formally defined by the axioms in table 4.4.

Definition 15 (Alphabet operators)**.** *The alphabet operators behave as follows:*

Symbol	Meaning
Γ_C	communication
∇_V	allow
ρ_R	rename
τ_I	hide

Table 4.3: The operators of ACP

- *The communication operator Γ has parameter C, a map from multi-actions to actions. The communication operator replaces all multi-actions m in C by $C(m)$.*
- *The allow operator ∇ has parameter V, a set of actions. The allow operator lets only actions a which are not in V execute. The only exception is $a = \tau$, which can execute regardless of V.*
- *The rename operator ρ has parameter R, a map from actions to actions. The rename operator replaces all actions a in R by $R(a)$.*
- *The hide operator τ has parameter I, a set of actions. The hide operator replaces all actions a in I by the unobservable, internal action τ.*

No τ action, nor any multi-action containing τ, ever occurs in C, V, R or I.

We define a unary operator SLS ('synchronization law set') with parameter Par. Par is a map from multi-actions —all of the same size n —to actions. The multi-actions in Par may map to τ, in contrast to parameter C of communication operator Γ, which is otherwise similar. We assume that SLS is applied to a set of n parallel processes.

Multi-actions in Par may contain, in some action places, the special symbol $\bullet$. When SLS_{Par} is applied to a process expression, a $\bullet$ symbol in place i denotes that the process in place i performs no action. SLS_{Par} is equivalent to $\tau(\rho(\nabla(\Gamma(P))))$, and it has behaviour defined more precisely in terms of the alphabet operators in Definition 16.

Definition 16 (Synchronization law set (SLS) operator)**.** *The behaviour of SLS_{Par} applied to parallel process expression S is equal to $\tau_I(\rho_R(\nabla_V(\Gamma_C(S))))$, with the following values for parameters C, V, R, and I:*

- *C maps*
  - *each multi-action m with $m \to a$, $a \neq \tau$ in Par to a hash action $h(a)$ which is unique for that multi-set not defined in P_S;*
  - *each multi-action m with $m \to \tau$ in Par to the reserved action int;*
- *V contains all hash actions from C and int;*
- *R maps each hash action $h(a)$ in V to a;*
- *$I = int$.*

Table 4.5 gives the resulting axioms for SLS_{Par}.

A process specification in pCRL format is called a process network if the new operators introduced for ACP only occur in the initialization expression. We assume a certain format for the initialization expression of such a process network, without loss of generality: we assume that I_S is a single vector of merged process variables $I_{S1} \| I_{S2} \| \ldots \| I_{Sn}$ called the initial variables, with a single SLS operator at the top level.

We also assume that the process variables in the initialization expression have disjoint process variable sets and action sets, that is, the transitive closure of the process definitions of initial

Label	Axiom
Communication	
C1	$\Gamma_C(\alpha) = C(\alpha)$
C2	$\Gamma_C(\delta) = \delta$
C3	$\Gamma_C(x + y) = \Gamma_C(x) + \Gamma_C(y)$
C4	$\Gamma_C(x \cdot y) = \Gamma_C(x) \cdot \Gamma_C(y)$
Allow	
V1	$\nabla_V(\alpha) = \alpha$ if $\alpha \in V \cup \{\tau\}$
V2	$\nabla_V(\alpha) = \delta$ if $\alpha \notin V \cup \{\tau\}$
V3	$\nabla_V(\delta) = \delta$
V4	$\nabla_V(x + y) = \nabla_V(x) + \nabla_(y)$
V5	$\nabla_V(x \cdot y)P = \nabla_V(x) \cdot \nabla_V(y)$
TV1	$\nabla_V(\nabla_W(x)) = \nabla_{V \cap W}(x)$
Rename	
R1	$\rho_R(\tau) = \tau$
R2	$\rho_R(a) = \beta$ if $\alpha \to \beta \in R$ for some β
R3	$\rho_R(a) = \alpha$ if $\alpha \to \beta \notin R$ for all β
R4	$\rho_R(\alpha \mid \beta) = \rho_R(\alpha) \mid \rho_R(\beta)$
R5	$\rho_R(\delta) = \delta$
R6	$\rho_R(x + y) = \rho_R(x) + \rho_R(y)$
R7	$\rho_R(x \cdot y) = \rho_R(x) \cdot \rho_R(y)$
Hide	
H1	$\tau_I(\tau) = \tau$
H2	$\tau_I(\alpha) = \tau$ if $\alpha \in I$
H3	$\tau_I(\alpha) = \alpha$ if $\alpha \notin I$
H4	$\tau_I(\alpha \mid \beta) = \tau_I(\alpha) \mid \tau_I(\beta)$
H5	$\tau_I(\delta) = \delta$
H6	$\tau_I(x + y) = \tau_I(x) + \tau_I(y)$
H7	$\tau_I(x \cdot y) = \tau_I(x) \cdot \tau_I(y)$
H10	$\tau_I(\tau'_I(x)) = \tau_{I \cup I'}(x)$

Table 4.4: The alphabet axioms

Label	Axiom
SLS1	$SLS_{Par}(\tau) = \tau$
SLS2	$SLS_{Par}(\alpha) = \tau$ if $\alpha \to \tau \in Par$
SLS3	$SLS_{Par}(\alpha) = \beta$ if $\alpha \to \beta \in Par$ for some $\beta \neq \tau$
SLS4	$SLS_{Par}(\alpha) = \delta$ if $\alpha \to \beta \notin Par$ for all β
SLS5	$SLS_{Par}(\delta) = \delta$
SLS6	$SLS_{Par}(x + y) = SLS_{Par}(x) + SLS_{Par}(y)$
SLS7	$SLS_{Par}(x \cdot y) = SLS_{Par}(x) \cdot SLS_{Par}(y)$

Table 4.5: The SLS axioms

variables $I_{S\,i}$ and $I_{S\,j}$ with $i \neq j$ do not contain the same process variables or actions (except τ and δ).

In the remainder of this thesis process networks are presented in a format which is formally defined in Definition 17.

Definition 17 (Process network). *A process network M is presented as $\langle \Pi_M, V_M \rangle$ with Π_M a vector of n BPA process expressions in pCRL format and V_M a synchronization law set containing multi-actions of size n.*

After complete distribution of the synchronization operator over and through the processes in the process vector Π_M we arrive at a single pCRL process expression: M's system process. Each state in M's system process corresponds to a system state of the process vector Π_M. Each action in the system process corresponds to a multi-action in the process vector Π_M.

4.3 Rule systems and application conditions

We can use an individual transformation rule to transform a system of parallel processes, as we did for single processes in chapter 3. To apply an individual transformation rule, we look for matches in all processes and replace each match by the corresponding right pattern.

However, we may also want to transform behaviour which transcends the level of single processes, like transforming the communication between two processes from synchronous to asynchronous. This requires the application of a system of multiple transformation rules at once, potentially changing the synchronization laws as well. The verification of such *rule systems* is more involved: any individual transformation rule may not preserve branching bisimilarity on a process, but together the involved transformation rules may still preserve branching bisimilarity.

A rule system is defined in Definition 18.

Definition 18 (Rule system). *A rule system $\Sigma = \langle R, \hat{V}, \hat{V}' \rangle$, with R a transformation rule set and $\hat{V}$ and $\hat{V}'$ synchronization law sets.*

In a rule system, synchronization law set $\hat{V}$ represents laws which must already be present in the process network and synchronization law set $\hat{V}'$ represents laws which are introduced by the transformation.

An application of a rule system to a process network can only take place if $\hat{V} \subset V$. The result of an application of a rule system is given in Definition 19.

Definition 19 (Network application). *The application of rule system $\Sigma = \langle R, \hat{V}, \hat{V}' \rangle$ to process network $M = \langle \Pi, M \rangle$ consists of two steps regarding the Π:*

1. *First for every combination of a process in Π and a transformation rule in R the largest possible set of matches is found.*

2. *Second each transformation rule is applied at all of its matches as per Definition 11 of transformation rule application.*

The synchronization laws in $\hat{V}' \setminus \hat{V}$ are added to the synchronization laws V of the process network.

To make reasoning about transformation of synchronizing behaviour easier, Definition 20 introduces a few restrictions on the application of rule systems. If and only if the combination of rule system Σ and process network M satisfies these conditions, then Σ may be applied to M.

Definition 20 (Synchronization uniformity). *A rule system Σ is called synchronization uniform with respect to a process network M if the following conditions hold:*

Universal applicability (L): *For all actions a in the definition $P_M(V)$ of some variable V in M which also occur in the multi-action of some transformation rule in R_Σ, if some synchronization law $\hat{V}_\Sigma$ contains a and at least one other action, then all variables V' in processes in vector Π_M whose definition $D_M(V')$ contains action a must be matched by some transformation rule in R_Σ;*

Universal applicability (R): *symmetrically for $\Sigma(M)$ and $\hat{V}_\sigma \cup \hat{V'}_\Sigma$.*

Completeness (L): *For all actions a in the definition $P_M(V)$ of some variable V in M which also occur in the multi-action of some transformation rule in R_Σ, if some synchronization law $\hat{V}_\Sigma$ contains both a and some action b, then there must be a multi-action of some transformation rule in R_Σ which contains some variable V with a definition $D_{R_\Sigma}(V)$ which contains action b;*

Completeness (R): *symmetrically for $\Sigma(M)$ and $\hat{V}_\sigma \cup \hat{V'}_\Sigma$.*

Synchronization: *For all actions a in the multi-action of some synchronization law $L \in \hat{V'}_\Sigma \backslash \hat{V}_\Sigma$, $a \notin A_M$.*

4.4 Pattern networks

When a rule system is applied to a process network, there may be many matches. It may happen that one rule matches multiple processes, or that multiple rules match the same process, or that a rule matches multiple times on the same process. The existence of so many matches complicates our proof. In the rest of this thesis we limit our reasoning to the *vector case* defined in Definition 21.

Definition 21 (Vector case)**.** *A process network $M = \langle \Pi, V \rangle$ is transformed by a rule system $\Sigma = \langle R, \hat{V}, \hat{V'} \rangle$, and for all i, $i \leq |\Pi|$, the left pattern of transformation rule $R[i]$ matches exactly once, namely on process $\Pi[i]$.*

In any rule system we require that the transformation rule set is confluent: any order of application of the rules should lead to the same result. Recent work discusses how to check confluence of rule systems efficiently for the LTS-based case [30]. Confluence of rule systems ensures that our results for the vector case which we examine here can be generalized to arbitrary matching.

The vector case gets its name from the fact that the transformation rule set of Σ can be interpreted as a vector. This is convenient for verification because we can now analyse the behaviour of the patterns of a rule system as if they were process networks by analysing its *pattern networks*, as defined in Definition 22.

Definition 22 (Pattern networks)**.** *For rule system $\Sigma = \langle R, \hat{V}, \hat{V'} \rangle$ with $|R| = n$:*

- *The left pattern network L_Σ is a process network $\langle \Pi_L, V_L \rangle$ with:*
  - *For all $i \leq n$, process $\Pi_{Li} = L_{R_i}$;*
  - *Synchronization law set $V_L = \hat{V}$.*
- *The right pattern network R_Σ is a process network $\langle \Pi_R, V_R \rangle$ with:*
  - *For all $i \leq n$, process $\Pi_{Ri} = R_{R_i}$;*
  - *Synchronization law set $V_R = \hat{V'}$.*

Like a regular process network, a pattern network is a parallel composition of processes under an SLS operator. As explained in Section 4.1, if multiple processes in a parallel composition can take actions, then the parallel composition can take a multi-action. And as we have seen in Section 4.2, the synchronization law set operator SLS maps multi-actions to single actions of the process network as a whole.

In the following sections it is sometimes convenient to view a process network as the single *network process* which results from the application of its synchronization law set to its process vector, factoring out the parallel composition operator. The process variables of the network process have no direct definition in the process network specification, but correspond to *variable vectors* whose multi-actions are composed by the synchronization law set into single actions of the network process.

Recall that a match is a map from pattern variables to process variables. For variable vectors we define the similar concept of *vector match* in Definition 23. In our proof of bisimulation preservation we reason about such vector matches.

Definition 23 (Vector match)**.** *Suppose that variable vector $\overline{V}$ of length n is part of a pattern network N, and variable vector S of length n is part of a process network M. If for all i, $i \leq n$ pattern i of N matches process network M, then $\overline{V}$ has a vector match m on S such that for each index i we have $m(\overline{V}[i]) = S[i]$.*

It may happen that some rule has a match, and it needs another rule to form a multi-action, but that other rule does not have a match. In that case a certain communication is impossible. We need to be able to reason about such unsuccessful communication in addition to successful communication. To reason about unsuccessful and successful communication we introduce introduces dependency relations D and $D*$ on rules in Definition 24.

Definition 24 (Dependency relations)**.** *In a rule system Σ, transformation rules T_1 and T_2 are related by* direct dependency relation *D if action a occurring in T_1 and action b occurring in T_2 occur together in a multi-action of a synchronization law in $\hat{V}_\Sigma \cup \hat{V}'_\Sigma$.*

The dependency relation *$D*$ is the transitive closure of D. The dependency relation can be used to partition the indices of a rule system's transformation rule vector into a* dependency set *partition $\mathbb{D}_\Sigma$.*

To prove the bisimulation preservation of a rule system Σ regardless of the process network to which it is applied, we investigate each subset I of each dependency set in $\mathbb{D}_\Sigma$. We filter the rule system by the dependency set, as defined in Definition 25.

Definition 25 (Filtering)**.** *Filtering the rule system $\Sigma = \langle R, \mathcal{V}, \mathcal{V}' \rangle$ by index set I turns it into filtered rule system $\Sigma_I = \langle R_I, \mathcal{V}_I, \mathcal{V}'_I \rangle$ where each element $R_{I\,j}$ with $j \notin I$ is equal to $\langle d, d \rangle$ where d is the* dummy process*:*

$A_d : ::= \{\}$

$P_d : ::=$

$\quad V_\delta \ = \delta$

$I_d : = V_\delta.$

and $\mathcal{V}_I$ and $\mathcal{V}'_I$ are equal to $\mathcal{V}$ and $\mathcal{V}'$, respectively, except that all laws whose multi-actions require an action from the process at index j are removed from both.

An optional second step is *shortening* a filtered rule system, as defined in Definition 26. We can shorten a process network without a change to the network process.

Definition 26 (Shortening)**.** *A process network (for example a filtered rule system) is shortened by removing every dummy process. For each such process at an index j, each synchronization law has a $\bullet$ as the jth element in its multi-action. These jth entries are removed as well.*

We take the left and right pattern networks L_I and R_I for the filtered process network of I, and check bisimulation between L_I and R_I. But when doing so we need to take into account the actions which enter and exit the pattern.

As explained in Definition 13, when we check bisimulation preservation on individual transformation rules, we extend these rules with a variable V_κ and κ-summands.

In the same way, when we check bisimulation between pattern networks, the individual pattern processes in the network are κ-extended. But when analyzing the pattern network we can not stop at checking the individual actions which can enter or exit a pattern: we need to take into account all multi-actions which may enter or exit the pattern. Therefore we introduce synchronization laws for multi-actions consisting of κ-actions in Definition 27.

Definition 27 (κ-synchronization)**.** *In a κ-extended pattern network N^κ each individual transformation rule is κ-extended as per Definition 13.*

Additionally the synchronization law set $\hat{V}$ of N^κ is extended with a set of synchronization laws ${V^\kappa}_I$ for each subset $I \in \mathbb{D}_N$ of the processes of N^κ.

For such a subset I the set ${V^\kappa}_I$ contains all multi-actions which consist of, for each process in Π_N, either one κ-action occurring in Π_N or the special symbol $\bullet$, except the multi-action consisting of only $\bullet$ symbols. ${V^\kappa}_I$ maps each such multi-action to a new unique κ-action.

When we know for every subset I of every dependency set in $\mathbb{D}_\Sigma$ whether it holds for its left and right pattern networks L_I and R_I that ${L_I}^\kappa \underline{\leftrightarrow}_{rb} {R_I}^\kappa$, we can prove that Σ is bisimulation preserving.

4.5 Preservation of branching bisimulation

We proceed to prove Proposition 2 which states that branching bisimulation is preserved by the application of rule systems with certain properties.

Proposition 2. *Suppose that we have a rule system Σ with R_Σ of size n and for all subsets I of elements of $\mathbb{D}_\Sigma$, ${L_I}^\kappa \underline{\leftrightarrow}_{rb} {R_I}^\kappa$.*

A synchronization uniform application of Σ on process network M at vector match m of left pattern network process L_Σ results in a process specification $\Sigma(M, m)$, where $M \underline{\leftrightarrow}_{rb} \Sigma(M, m)$ and right pattern network process R_Σ has match m' on $T(S, m)$.

Proof. We give a proof sketch.

Similarly to the situation in the proof of Proposition 1 we seek to preserve that $I_M \underline{\leftrightarrow}_{rb} I_{\Sigma(M,m)}$ (the initial variable vectors of the network processes are related) and all behaviour of transitions of M's and $\Sigma(M, m)$'s solutions is preserved.

To see that $I_M \underline{\leftrightarrow}_{rb} I_{\Sigma(M,m)}$, note that the initial state of a network process is the vector of all initial variables of its sub-processes. All of these initial states are matched on by glue variables, and so they are preserved by the transformation, so $I_M = I_{\Sigma(M,m)}$.

We now show that all behaviour of transitions of M's and $\Sigma(M, m)$'s solutions is preserved. We build a relation C between M and $\Sigma(M, m)$ which will be our proposed branching bisimulation relation. First we define relations A_i between all unmatched variables in all processes and themselves. Next, recall that we have established a branching bisimulation relation $\hat{B}_i$ between each pair of pattern variables $V_1 = {L_\Sigma}_i^\kappa$ and $V_2 = {R_\Sigma}_i^\kappa$. We define a relation B_i between each pair of process variables $m(V_1)$ and $m(V_2)$. Each variable of each process in M is in either some A_i or some B_i.

We define relation C' as $\cup_{i \in I}(A_i \cup B_i)$. We then define C as the relation between vectors which maps every vector $\overline{V} = [V_0, V_1 \ldots V_n]$ to $C(\overline{V}) = [C'(V_0), C'(V_1) \ldots C'(V_n)]$. In the remainder of this proof we establish that C is indeed a rooted branching bisimulation relation.

We show that for every $i \leq n$, $I_{M_i} \underline{\leftrightarrow}_{rb} I_{T(M_i,m)}$: $I_{T(M_i,m)} = I_{M_i}$ holds by Definition 11, so if I_{M_i} is not matched then $I_{M_i} \in A$, and if I_{M_i} is matched then it is a glue variable, and $I_{M_i} \in \hat{B}_i$. Lemma 2 ensures that rootedness is not an issue. Since for every $i \leq n$, $I_{M_i} \underline{\leftrightarrow}_{rb} I_{T(M_i,m)}$, we also have $I_M \underline{\leftrightarrow}_{rb} I_{T(M,m)}$: the initial variable vectors of the network processes are related.

Next we show that all behaviour of transitions of M's and $T(M,m)$'s solutions is preserved. Consider first an arbitrary variable vector $\overline{V}$ in M. For each multi-action $\overline{a}$ of $\overline{V}$ with a synchronization law $\hat{V}_i = (\overline{a} \rightarrow b)$, we distinguish two cases:

- If there exists a matched $V \in \overline{V}$ with a summand $a \cdot W$ with action $a \in \overline{a}$, then by synchronization uniformity (Definition 20) for all summands $a' \cdot W_i$ with action $a_i \in \overline{a}$ the behaviour of a' must be preserved. So $\overline{V}$ is matched completely and each element is in some B_i. From B we can find a related variable vector $\overline{V'}$ which shows that $\overline{V}$ is related to some bisimilar element by C.
- If there exists no matched $V \in \overline{V}$ with a summand $a' \cdot W_i$ with action $a' \in \overline{a}$, then we can distinguish three cases for element V_i of $\overline{V}$:
  - V_i is related to itself by relation A_i. W_i is also in A_i and related to itself. The summand is not involved in the transformation, so its behaviour is preserved.
  - V_i is related to itself by relation A_i. $m(W_i)$ is in $\hat{B}_i$ and it is a glue variable, so W is related to itself. The summand is not involved in the transformation, so its behaviour is preserved.
  - V_i is in some B_i and is related to some V_i', but it has no matched summands $a' \cdot W_i$ with $a' \in \overline{a}$. Either V_i has no summands or it is an exit variable. If V_i has no summands there is no behaviour to be preserved. If $m(V_i)$ is an exit variable then V is related to itself by B_i and variable W is in A_i and related to itself. The summand is not involved in the transformation, so its behaviour is preserved.

We have now also established that all behaviour of the process network M is preserved by C. Since the definition of a match is symmetric, we can apply the same reasoning to the existence of all behaviour of resulting process specification $T(M,m)$ in M. So C is indeed a rooted branching bisimulation relation. □

Remark 2 (Contributions). The specification format used in this chapter is taken from the literature, except the SLS operator, which is new. We lifted the correctness check from the LTS level to the process algebra level.

Previously de Putter discovered the problem of *non-cascading* rule systems [25]. Unpublished work by de Putter and Wijs establishes an approach to eliminating this problem, which involves V_κ and κ-synchronization. This is the first work in which that approach is used. The author contributed to discussions leading to improvements in the approach.

Previous work explicitly includes the concept of *admissibility* of rule systems, which ensures certain properties relating to τ-actions. We require the same properties, but our use of the hiding alphabet operator naturally ensures that rule systems are admissible.

To describe the expressiveness of the method so far, Definition 28 introduces the concept of pairwise bisimilarity.

Definition 28 (Pairwise branching bisimilar). *A pair of process networks is pairwise (rooted) branching bisimilar if every corresponding element pair is (rooted) branching bisimilar under the network's synchronization laws and the network processes are also (rooted) branching bisimilar.*

For each pairwise rooted branching bisimilar pair of process networks A and B there exists a rule system R such that $T(A) = B$. This is not very expressive yet. We will gain much more expressiveness in the next chapter.

4.6 Example

Example 2. We define a transformation rule $T2_1 = \langle L2_1, R2_1 \rangle$ as follows:

$L2_1 : A ::= \{a, s1\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1;$
$V_1 = a \cdot V_2;$
$V_2 = \kappa_2 \cdot V_\kappa$

$I ::= V_{Synch}$

$R2_1 : A ::= \{a, s1\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1;$
$V_1 = a \cdot V_2;$
$V_2 = s1 \cdot V_2 + \kappa_2 \cdot V_\kappa$

$I ::= V_{Synch}$

And we define a transformation rule $T2_2 = \langle L2_2, R2_2 \rangle$ as follows:

$L2_2 : A ::= \{b, s2\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1;$
$V_1 = b \cdot V_2;$
$V_2 = \kappa_2 \cdot V_\kappa$

$I ::= V_{Synch}$

$R2_2 : A ::= \{b, s2\}$

$P ::=$

$V_\kappa = \kappa_1 \cdot V_1;$
$V_1 = b \cdot V_2;$
$V_2 = s2 \cdot V_2 + \kappa_2 \cdot V_\kappa$

$I ::= R_{Synch}$

Both transformation rules simply add a τ-loop to variable V_2. We also define two sets of synchronization laws. The first synchronization law set $\mathcal{V} = \{\}$: the empty set. The second synchronization law set $\mathcal{V}' = \{(\langle s1|s2\rangle, \tau)\}$.

We can now create a rule system. Rule system $\Sigma_1 = ([R2_1, R2_2], \mathcal{V}, \mathcal{V}')$. Informally, rule system Σ_1 can be applied to a process network $M1$ which consists of two parallel processes and a synchronization law set which includes the laws in V. It applies transformation rule $R2_1$ to the first process $M1_1$ and transformation rule $R2_2$ to the second process $M1_2$ and it adds $\mathcal{V}'$ to the synchronization law set of $M1$.

We check for each individual transformation rule of Σ_1 ($R2_1$ and $R2_2$, in this case) whether the left and right patterns of the transformation rule are bisimilar.

We then combine the parallel composition of the left patterns of all transformation rules with $\hat{V}_{\Sigma_1}$, which is $\mathcal{V}$, into a process network C_1. we similarly combine the parallel composition of the right patterns of all transformation rules with $\hat{V'}_{\Sigma_1}$, which is $\mathcal{V} \cup \mathcal{V}'$, into a process network C_2. The network processes of C_1 and C_2 look as follows:

$C_1 : A ::= \{a, b, s1, s2\}$

$P ::=$

$V_{\kappa 1} = \kappa_{21} \cdot V_{11};$
$V_{11} = a \cdot V_{21};$
$V_{21} = \kappa_{21} \cdot V_{\kappa 1}$
$V_{\kappa 2} = \kappa_{12} \cdot V_{12};$

$$
\begin{aligned}
&V_{12} = b \cdot V_{22};\\
&V_{22} = \kappa_{22} \cdot V_{\kappa 2}\\
I ::= &V_{\kappa 1} || V_{\kappa 2}
\end{aligned}
$$

$$
\begin{aligned}
C_2 :\ A ::= &\{a, b, s1, s2\}\\
P ::= &\\
&V_{\kappa 1} = \kappa_1 \cdot V_{11};\\
&V_{11} = a \cdot V_{21};\\
&V_{21} = s1 \cdot V_{21} + \kappa_2 \cdot V_{\kappa 1}\\
&V_{\kappa 2} = \kappa_{12} \cdot V_{12};\\
&V_{12} = b \cdot V_{22};\\
&V_{22} = s2 \cdot V_{22} + \kappa_2 \cdot V_{\kappa 2}\\
I ::= &SLS(\{\langle s1|s2\rangle \to \tau\}, V_{\kappa 1} || V_{\kappa 2})
\end{aligned}
$$

We check whether C_1 is bisimilar to C_2, which is possible because both are in parallel pCRL format. All checks are positive. After finding out that the rule systems do indeed preserve bisimilarity, we can apply rule system Σ_1 to process networks for which the application is synchronization uniform. For example, consider the process network $M2$:

$$
\begin{aligned}
M2 :\ A ::= &\{a\}\\
P ::= &\\
&M2_1 = a \cdot P_1;\\
&P_1 = c\\
&M2_1 = b \cdot P_2;\\
&P_2 = c\\
I ::= &SLS(\{\}, M2_1 \| M2_2)
\end{aligned}
$$

All synchronization uniformity conditions hold for this combination of rule system and process network, so we can apply rule system Σ_1 to process network $M2$ to get the following transformed process network $M2'$:

$$
\begin{aligned}
M2' :\ A ::= &\{a\}\\
P ::= &\\
&M2_1 = a \cdot P_1;\\
&P_1 = s1 \cdot P_1 + c \cdot P_3\\
&M2_1 = b \cdot P_2;\\
&P_2 = s2 \cdot P_2 + c \cdot P_3\\
I ::= &SLS(\{\langle s1|s2\rangle \to \tau\}, M2_1 \| M2_2)
\end{aligned}
$$

The bisimilarity check on rule system Σ_1 ensures that the original process network $M2$ and the resulting process network $M2'$ are bisimilar.

Chapter 5

Extending the transformation method

In the past chapters we have established that the existing refinement method can be adapted for use in a process algebra setting.

In this chapter we extend the existing refinement method. We add signature elements with data and extend the specification format again. We define how the refinement method can be applied to systems with data and to irregular specifications. We also propose extensions to the refinement method with transformations which change the number of processes in a network.

5.1 Systems with data

In this chapter we work with a yet greater part of the mCRL2 syntax, which we will refer to as 'systems with data'. We add signature elements and operators for transforming systems with data: parameters, generalized choice (sum), and conditions. This means that we use almost all of the mCRL2 language, notably excluding signature elements for modeling time. Timed processes can be modeled using the data elements which we introduce here, but without specialized signature elements these cannot be analyzed efficiently by the toolset.

To transform systems with data, we need data types. mCRL2 includes a number of data types by default: boolean, positive, natural, integer, real, structured data types, function data types, lists, sets and bags. We assume the existence of these data types for our systems with data as well. These data types have their usual (for example mathematical) operators, which we do not list. More data types can be defined by the user, but that is beyond the scope of this project.

Symbol	Meaning
Σ	sum (generalized choice)
$\rightarrow$ ($\diamond$)	condition

Table 5.1: The operators of systems with data

5.1.1 Data parameters

In systems with data, process variables and actions may have parameters. Every occurrence of the same action or process variable has the same number of data parameters of the same type. The number and types of data parameters of an action or process variable of process specification S are specified in action set A_S and process variable set V_S, respectively. We represent data variables by lowercase letters and their types by uppercase letters.

When a process variable is defined as having data parameters, that process variable binds data variables of the corresponding types in its process definition, which means that occurrences of actions and process variables in process definitions may have data parameters which consist not of the typed data parameter specifications used in A_S and V_S but of data expressions. Data expressions which occur as parameters of actions and process variables in process definitions can only contain bound data variables and must be computable. Evaluating data expression parameters must result in a data value of the correct type for the containing action or process.

Data parameters can be used to more easily model systems in which data variables play a central role. Whereas previously each value for the data variables would have to be represented as a process variable, data can now be represented in a compact way. Without data parameters the size of regular specifications would rise exponentially with the number of finite-domain data variables in the model, the size of a specification now rises only linearly. Data variables also allow the description of systems with infinite-domain data variables in finite specifications.

In systems with data, all alphabet operators preserve data parameters. The communication operator Γ takes account of data: two actions may only communicate if their data variable has the same value. Synchronization laws do not contain data parameters, but a vector of actions can only form a multi-action if all actions in the vector have the same data values.

In Definition 2 we defined branching bisimilarity through simulation. In systems with data, an action only simulates another if these actions have the same number and type of parameters, with the same values.

5.1.2 Generalized choice operator

Data parameters allow the use of data in a system. But data also has to enter the system. Examples of data entering a system include situations where a process receives data from another process inside or outside its network, or when a process makes an unpredictable observation or nondeterministic choice. To model data entering a system we use the generalized choice operator Σ.

Systems with data may contain the unary generalized choice operator $\Sigma_{d:D}p$ with d a data variable, D a data type, and p a process expression. $\Sigma_{d:D}p$ allows a choice of any $e \in D$ for data variable d in sub-expression p. As its name implies, the generalized choice operator can be viewed as a generalization of choice operator $+$.

Just like a process variable, the Σ operator binds occurrences of its data parameter d in its subexpressions. Occurrences of data variables in expressions may have only one binding element, either a process variable or a Σ-operator. Operator Σ distributes over all alphabet operators, as is evident from the axioms in Table 5.2.

Label	Axiom
SUM1	$\Sigma_{d:D}x = x$
SUM3	$\Sigma_{d:D}X(d) = X(e) + \Sigma_{d:D}X(d)$ for some $e \in D$
SUM4	$\Sigma_{d:D}(X(d) + Y(d)) = \Sigma_{d:D}X(d) + \Sigma_{d:D}Y(d)$
SUM5	$(\Sigma_{d:D}X(d)) \cdot y = \Sigma_{d:D}X(d) \cdot y$
LM5	$(\Sigma_{d:D}X(d)) \between y = \Sigma_{d:D}X(d) \between y$
S8	$(\Sigma_{d:D}X(d)) \mid y = \Sigma_{d:D}X(d) \mid y$
C5	$\Gamma_C(\Sigma_{d:D}X(d)) = \Sigma_{d:D}\Gamma_C(X(d))$
V6	$\nabla_V(\Sigma_{d:D}X(d)) = \Sigma_{d:D}\nabla_V(X(d))$
R8	$\rho_R(\Sigma_{d:D}X(d)) = \Sigma_{d:D}\rho_R(X(d))$
H8	$\tau_I(\Sigma_{d:D}X(d)) = \Sigma_{d:D}\tau_I(X(d))$
SLS8	$SLS_{Par}(\Sigma_{d:D}X(d)) = \Sigma_{d:D}SLS_{Par}(X(d))$

Table 5.2: The Σ axioms

5.1.3 Conditional operator

So far we have introduced signature elements and operators which allow systems to contain data, to change data values and to accept new data values. But we also need a way to let the data of a system influence its control flow. To that end we introduce the conditional operator.

Systems with data may contain the conditional operator $c(d_1 \dots d_n) \rightarrow p \diamond q$ with c a boolean expression over bound variables $d_1 \dots d_n$ and p, q process expressions. If c is equivalent to true then expression $c \rightarrow p \diamond q$ is equivalent to p, else the expression is equivalent to q. We give axioms to this effect for the conditional operators in Table 5.3. For easy rewriting without an explicit treatment of data we give an axiom $ELSE$ which allows one to remove the $\diamond$ ('else') symbol.

Label	Axiom
Cond1	$true \rightarrow x \diamond y = x$
Cond2	$false \rightarrow x \diamond y = y$
THEN	$c \rightarrow x = c \rightarrow x \diamond \delta$
ELSE	$c \rightarrow x \diamond y = (c \rightarrow x) + ((\neg c) \rightarrow y)$

Table 5.3: The conditional axioms

5.1.4 Specification format

In addition to requiring that the expressions which define sets of parallel processes with data conform to the signature of our systems with data, we also require that sets of parallel processes are specified in the parallel pCRL format with data, which is an extension of the parallel pCRL format, as defined in Definition 14. Note that where the parallel pCRL format as described in the literature allows for the definition of one's own data types, we do not go into that in this thesis.

Definition 29 (Parallel pCRL format with data)**.** *A system with data specification S in parallel pCRL format with data is a tuple $\langle A_S, V_S, P_S, I_S \rangle$ with*

Action set A_S. *Each element a of A has a set number n of data parameters. For each data parameter i, $i \leq n$ the definition of action a contains name v_i and type t_i, specified in the form $a(v_1 : t_n \dots v_n : t_n)$. The unobservable, internal action τ and the deadlock δ are always part of the action set, have no parameters, and do not need to be specified;*

Process variable set V_S. *Each element V of V_S has a set number n of data parameters. For each data parameter i, $i \leq n$ the definition of V contains name v_i and type t_i, specified in the form $V(v_1 : t_n \dots v_n : t_n)$. V_S contains every process variable defined in P_S;*

Process definition set P_S. *A set of definitions which can be viewed as a function from elements of V_S to definitions in the form of process expressions. Only BPA operators and data operators, but no other ACP operators including no alphabet operators occur in the definitions;*

Initialization expression I_S, *which consists of a process expression.*

Note that like the more basic parallel pCRL format from Chapter 4, the parallel pCRL with data allows irregular processes. When working in the parallel pCRL with data format, we allow processes and transformation rule patterns to have an irregular form.

It has been established in previous work that any process network specification which contains only computable data expressions and a finite set of actions can be defined using a certain kind of regular form called a *Linear Process Specification* (LPS) with data which contains only computable functions over the natural numbers [24].

Such LPS with data contain only process variable definitions with, for process variable $V(d : D)$ summands of the form $\Sigma_{e:E} c(d, e) - > a(f(d, e)) \cdot V'(g(d, e))$ with f, g arbitrary functions, and c a boolean function, over data parameters d, e.

Since we require our process network specifications to be computable, we can assume regularity in our bisimulation preservation proofs even though we allow the specification of irregular systems and transformation rule patterns.

5.2 Transformation rules with data

When assessing bisimilarity in systems with data, we take into account the fact that the names which we use for data parameters are not visible in the behaviour of a process network: only data variables are visible. We can use the invisibility of data parameter names to make rule systems more broadly applicable: we define a form of α-conversion on data variables.

α-conversion involves replacing variable names in a specification by free variable names, that is, variable names not already used elsewhere in the specification. We define α-conversion more precisely in Definition 30.

Definition 30 (α-conversion)**.** *The α-conversion of a process network M is the result of applying some map A to M, replacing all variable names d by $A(d)$. The range A can not contain any variables already present in M before the α-conversion.*

The α-conversion of a process network M by map A is bisimilar to M itself. If we convert all patterns of a bisimulation preserving rule system by the same map A, the result is another bisimulation preserving rule system. Therefore we identify expressions and action or process variables which are equal modulo α-conversion.

When we take equivalence of the new operators modulo α-conversion into account, the following definitions can be used with systems with data in exactly the same way as they appear in Chapter 4 about ACP:

- Definition 18: Rule system
- Definition 19: Network application
- Definition 20: Synchronization uniformity
- Definition 21 Vector case
- Definition 22: Pattern networks
- Definition 23: Vector match
- Definition 24: Dependency relation
- Definition 25: Filtering
- Definition 26 Shortening
- Definition 27: κ-synchronization

The only addition of note is that κ-actions carry the parameters of the variable in whose definition they appear.

5.3 Preservation of branching bisimulation

In Chapter 4 we proved the correctness of a bisimulation preservation check for rule systems in ACP. In the literature there is some discussion on the correct way to model the behaviour of quantification and binding by sums, see for example the thorough treatment of this topic by Luttik [22].

We view network definitions with data as an abbreviation of a set of network specifications, one for each legal assignment of values to the data parameters in the initial expression. Note that

the size of the set of specifications which is abbreviated by some network specification with data may in fact be infinite.

Definition 31 describes a mapping which expands network specifications with data to the sets of network specifications which they abbreviate. We only discuss the case where all process variable definitions are regular, which we can do without loss of generality given our observation in the previous section that irregular network specifications can be reduced to regular ones. We only give the case where each action or process variable has a single data parameter. The generalization is obvious.

Definition 31 (Data expansion)**.** *A regular network specification in systems with data can be mapped to a (possibly infinite) expanded network specification (without data) by the following* expansion *mapping exp:*

- *Each action $a(d:D)$ maps to an action a_d.*
- *Each process variable $V(d:D)$ maps to a process variable V_d.*
- *Each defining expression $P_M(V)$ of process variable $V(d:D)$ consists of a number of summands of the form $\Sigma_{e:E} c(d,e) -> a(f(d,e)) \cdot V'(g(d,e))$ with f, g arbitrary functions over data parameters d, e.*

  In the expanded definition *$exp(V(d:D))$ each such summand s is replaced by at most $|D| \times |E|$ summands $P_M(V)_{d:D,e:E}$, one for each combination of d and e for which $c(d,e)$ holds, where $P_M(V)_{d:D,e:E} = a_{f(d,e)} \cdot V'_{g(d,e)}$.*
- *Each process variable $V(d)$ in the initial expression maps to a process variable V_d.*

Our interpretation as abbreviations with an expansion defined through the expansion mapping leads automatically to the definition of rooted branching bisimulation for systems with data which we give in Definition 32.

Definition 32 (Bisimulation with data)**.** *For two network specifications M_1 and M_2 with data it holds that $M_1 \underline{\leftrightarrow}_{rb} M_2$ if and only if there exists a rooted branching bisimulation relation between the process variables in their expanded network specifications such that $exp(M_1) \underline{\leftrightarrow}_{rb} exp(M_2)$.*

Now that we have an abbreviation mapping and a definition of rooted branching bisimulation with data we can prove Proposition 3.

Proposition 3. *Suppose that we have a rule system Σ with data, with R_Σ of size n and for all subsets I of elements of $\mathbb{D}_\Sigma$, ${L_I}^\kappa \underline{\leftrightarrow}_{rb} {R_I}^\kappa$.*

An application of Σ on process network M with data at vector match m of left pattern network process L_Σ results in a process specification $\Sigma(M,m)$, where $M \underline{\leftrightarrow}_{rb} \Sigma(M,m)$ and right pattern network process R_Σ has match m' on $T(S,m)$.

Proof. We only need to modify the proof of Proposition 2 somewhat to update the relations from which our rooted branching bisimulation is composed. Recall that process variable vector relation C was built from process variable relation C', which was equal to $A_i \cup B_i$. We show that we can still build relations A' and B' such that the proof works.

- Relation A_i related those process variables which do not change. To show that we can construct such a relationship A_i between unmatched process variables with data, we need to show that two equal process variables with data have rooted branching bisimilar expansions.

  If two process definitions with data $P_{M\,i}$ and $P_{M\,j}$ are exactly equal then trivially $exp(P_{M\,i}) = exp(P_{M\,j})$. So relation A_i can be constructed for systems with data.

- Relation B_i related those process variables $m(V_i)$ and $m(V_j)$ for which $V_i\hat{(}B_i)V_j$ where B_i was the bisimulation assumed in the proposition. To show that we can construct such a relationship B_i between matched process variables with data, we need to show that two process variables $m(V_i), m(V_j)$ with data, matched by bisimilar pattern variables with data V_i, V_j with $V_i\hat{(}B_i)V_j$, it holds that $m(V_i)\underline{\leftrightarrow}_{rb}m(V_j)$

  If $V_i\hat{(}B_i)V_j$ then by Definition 32 of data bisimulation with data we have $exp(V_i)\underline{\leftrightarrow}_{rb}exp(V_j)$.

  If in match m left pattern variable V matches process variable $m(V)$, then by Definition 10 of a match, if the transformation rule is applied on every matched variable, then $P_{M_i}(m(V)) = P_{L_{R_i}}(V)$ modulo associativity and commutativity of operators. Since no associativity or commutativity plays a role inside a summand, such variables V and V' have the same summands sets, so by Definition 31 of data expansion we have $exp(V) = exp(m(V))$.

  Since we have $exp(V_i)\underline{\leftrightarrow}_{rb}exp(V_j)$, we have $exp(m(V_i))\underline{\leftrightarrow}_{rb}exp(m(V_j))$. By Definition 32 of data bisimulation also have $V_i\underline{\leftrightarrow}_{rb}V_j$. So relation B_i can be constructed for systems with data, too.

We can construct suitable relations A and B, so we can construct a suitable relation C' and therefore C as well. With these relations we can adapt the proof of Proposition 2 to prove Proposition 3. □

5.4 Compositional refinement

Compositional refinement is the transformation of a network in a modular way: adding and removing synchronization laws, adding and removing processes and transforming part of a system in isolation. We briefly describe a few ways of refining systems compositionally.

5.4.1 Bidirectionality

So far we have viewed transformation rules as one-way affairs, but because the proofs of correctness of transformation rules are based on equivalence we can also apply any transformation rule the other way. We call this *reverse application*.

Using a reverse application, we can remove synchronization laws. The synchronization condition of synchronization uniformity is reversed in such a case: the multi-actions of removed synchronization laws cannot contain actions which are still present in the resulting system after application of the transformation rule.

5.4.2 Network expansion

We have defined the operation of shortening a filtered rule system in Definition 26. We can also apply shortening on normal process networks outside the context of filtering. That way we can remove processes without functionality (which are equivalent to the dummy process).

Additionally, if we apply shortening in reverse, we get an *expansion* operation in Definition 33. In expansion a new dummy process is added to the process vector, and the multi-actions in the synchronization law set are expanded by $\bullet$ symbols accordingly. No bisimilarity checks are required to expand a process network. The new dummy process can later be transformed into a nontrivial process. This way we can add any number of new processes to a network.

Definition 33 (Expanding)**.** *A process network is expanded by adding a number of dummy processes. For each such new process at an index j, a $\bullet$ action is inserted into each remaining synchronization law as the jth element in its multi-action.*

5.4.3 Splitting rules and nested process networks

We observed earlier in Section 4.2 that the process vectors in initialization expressions of process networks could be nested, and we chose to require that no nesting occurred. We have since observed that each process network is equivalent to a network process. Using the latter observation, Definition 34 defines a *splitting rule* which has a process P as its left side and a process network M of which P is the system process as its right side. Splitting rules can be used to split an individual process of a process network M into a new process network M' nested inside M.

Definition 34 (Splitting rule)**.** *A splitting rule is of the form* $\langle P, M \rangle$ *with* $P \underline{\leftrightarrow}_{rb} P_M$*, where* P_M *is the network process of* M*.*

A splitting rule matches a process if P matches the whole process. An application of a splitting rule replaces P by M. The reverse application of a splitting rule merges a process network into a single process.

5.4.4 Lifting rules and subvector transformation

Given the existence of nested process networks an obvious further question is how to move processes between different nesting levels inside a tree of nested networks. To move processes between nesting levels, Definition 35 defines a *lowering rule*. Definitions 36 and 37 define what matching and application mean for lowering rules.

Definition 35 (lowering rule)**.** *A lowering rule is of the form* $N = \langle P, V \rangle$*, with* N *a process network with:*

- P *a process vector of size* n*;*
- V *a synchronization law set of size* n *in which all multi-actions and resulting actions are unique.*

Definition 36 (lowering match)**.** *A lowering rule has a lowering match on a process network* $M = \langle \Pi, V \rangle$ *at index* i *if:*

- *each process* Π_{i+j} *matches* P_j *for all* $j < n$*;*
- *for each synchronization law* $k \in V$*, subvector* $\overline{m_k} = V_{k\,i \ldots i+n-1}$ *is present as a multi-action in* V'*, unless* $\overline{m_k}$ *consists of only* $\bullet$ *symbols.*

Definition 37 (lowering application)**.** *An application of a lowering rule consists of:*

- *replacing the subvector* Π_{i+j} *of* Π *by a subvector with one element: the system process of* N*;*
- *for each synchronization law* $k \in V$*, replacing subvector* $\overline{m_k} = V_{k\,i \ldots i+n-1}$ *by:*
  - *if* $\overline{m_k}$ *does not consist of only* $\bullet$ *symbols: a subvector with one element, namely the resulting action of* $V'(\overline{m_k})$*;*
  - *if* $\overline{m_k}$ *consists of only* $\bullet$ *symbols: a* $\bullet$ *symbol.*

The reverse application of a lowering rule lifts a subvector of a process network's process vector to a higher nesting level. This is only possible if the lower rule has a *splitting match* and the result is a *splitting application*, as defined in Definitions 38 and 39, respectively.

Definition 38 (lifting match)**.** *A lowering rule has a lifting match on a process network* $M = \langle \Pi, V \rangle$ *at index* i *if:*

- *process* Π_i *matches* P*;*

- *for each synchronization law $k \in V$, element $m_k = V_{k\,i}$ is present as a resulting action in V', unless m_k is a $\bullet$ symbol.*

Definition 39 (lifting application)**.** *A reverse application of a lowering rule (lifting application) consists of:*

- *replacing the subvector of Π consisting of the single process Π_i by a subvector equal to P;*
- *for each synchronization law $k \in V$, replacing one-element subvector $m_k = V_{k\,i}$ by:*
  - *if m_k is not a $\bullet$ symbol: a subvector equal to the multi-action $V'^{-1}(m_k)$;*
  - *if m_k is a $\bullet$ symbol: a vector of length n of $\bullet$ symbols.*

By first splitting a process and then lifting the resulting process network, we split the behaviour of a process over multiple parallel processes inside the same process network. This was not generally possible earlier.

If we want to transform only a subvector of the process network in isolation, we can first lower the subvector, then transform it, and then lift it again. This saves us from taking unchanged processes into account when transforming only part of a process network.

5.5 Contributions

The specification format with data used in this chapter is taken from the literature. We extended the transformation method with data, irregularity and compositional refinement. The bisimulation preservation check correctness proof (sketch) for the extended method is new.

Reverse application is also new in this thesis.

Previous work by Wijs discusses *maximal hiding*, which is a way to greatly reduce the verification effort for a rule system in practice by identifying a set of actions which are irrelevant to the properties which need to be maintained, and then hiding that *hiding set* [23]. Work by Wijs on compositional development [29] defines concepts for compositional development which allow one to apply maximal hiding to subsystems under transformation. The compositional development rules in our extended refinement method do not accomodate maximal hiding on lowered subnetworks, which means that it is less powerful in that regard.

Finally we prove proposition 4 about the expressiveness of the extended method.

Proposition 4. *For each bisimilar pair of nested process networks P and Q there exists a series of transformation rules S such that $S(P) = Q$.*

Proof. We apply the following transformation rules.

- Let $R1_P$ be a series of lifting rules which transforms P into a single, unnested process network P_1;
- let $R1_Q$ be a series of lifting rules which transforms Q into a single, unnested process network Q_1;
- let $R2_P$ be the splitting rule which transforms P_2 into its system process P_1;
- let $R2_Q$ be the splitting rule which transforms Q_2 into its system process Q_1.

Note that $P \underline{\leftrightarrow}_{rb} P_1 \underline{\leftrightarrow}_{rb} P_2$ and $Q \underline{\leftrightarrow}_{rb} Q_1 \underline{\leftrightarrow}_{rb} Q_2$. Since $P \underline{\leftrightarrow}_{rb} Q$ by assumption, we have $P_2 \underline{\leftrightarrow}_{rb} Q_2$ and since P_2 and Q_2 are single processes, there exists a rule R3 which transforms P_2 into Q_2.

We can apply transformation rules in reverse. Let $Rev(R2_Q)$ be $R2_Q$ applied in reverse. Let $Rev(R1_Q)$ be $R1_Q$ applied in reverse. Then we can pick for S the series of transformation rules $R1_P + [R2_P] + R3 + [Rev(R2_Q)] + Rev(R1_Q)$. □

5.6 Example

We give a producer-consumer example adapted from an example which was previously presented in LTS form in work by Wijs [29].

Example 3. We define a synchronization law set V_3 as follows:
$V_3 = \{(\langle s1|r1\rangle, \tau), (\langle r2|s2\rangle, \tau)\}$
And we define a process network $M3$ as follows:

$M3:\ A ::= \{produce, s1, consume, s2 : Int, r1, r2\}$
$P ::=$
$C_0 = \Sigma_{n:Int} \cdot produce(n) \cdot C_1(n);$
$C_1(n) = s1(n) \cdot r2 \cdot C_0$
$S_0 = \Sigma_{n:Int} \cdot r1(n) \cdot (n\rangle 0) \rightarrow s2 \cdot S_2(n);$
$S_2(n) = consume(n) \cdot S_0$
$I ::= SLS(V3, C_0 \| S_0)$

Process C_0 produces an integer n and sends it to process S_0, which checks whether n is larger than 0, and if so consumes n. Note the use of data and the irregular form of some of the process expressions in C_0 and S_0.

Suppose that we want to insert channels between the two existing components of $M3$, through which their communications are routed. To that end, we define four transformation rules $T3_1 \ldots T3_4$ as follows:

$L3_1:\ A ::= \{s1, s11 : Int, r2, r22\}$
$P ::=$
$V_\kappa(n) = \kappa_0(n) \cdot V_0(n)$
$V_0(n) = s1(n) \cdot r2 \cdot V_1;$
$V_1 = \kappa_1 \cdot V_\kappa$
$I ::= V_\kappa(n)$

$R3_1:\ A ::= \{s1, s11 : Int, r2, r22\}$
$P ::=$
$V_\kappa(n) = \kappa_0(n) \cdot V_0(n)$
$V_0(n) = s11(n) \cdot r22 \cdot V_1;$
$V_1 = \kappa_1 \cdot V_\kappa$
$I ::= V_\kappa(n)$

$L3_2:\ A ::= \{s1, s11 : Int, r2, r22\}$
$P ::=$
$V_\kappa(n) = \kappa_0(n) \cdot V_0(n)$
$V_0(n) = s1(n) \cdot r2 \cdot V_1;$
$V_1 = \kappa_1 \cdot V_\kappa$
$I ::= V_\kappa(n)$

$R3_2:\ A ::= \{s1, s11 : Int, r2, r22\}$
$P ::=$
$V_\kappa(n) = \kappa_0(n) \cdot V_0(n)$
$V_0(n) = s11(n) \cdot r22 \cdot V_1;$
$V_1 = \kappa_1 \cdot V_1$

$$I ::= V_\kappa(n)$$

$L3_3$: $A ::= \{s12, r11 : Int\}$

$$P ::=$$
$$V_\kappa = \kappa_0 \cdot V_0$$
$$V_0 = \kappa_1 \cdot V_\kappa$$
$$I ::= V_\kappa$$

$R3_3$: $A ::= \{s12, r11 : Int\}$

$$P ::=$$
$$V_\kappa = \kappa_0 \cdot V_0$$
$$V_0 = \Sigma_{n:Int} \cdot r11(n) \cdot s12(n) \cdot V_0 + \kappa_1 \cdot V_\kappa$$
$$I ::= V_\kappa$$

$L3_4$: $A ::= \{s22, r21\}$

$$P ::=$$
$$V_\kappa = \kappa_0 \cdot V_0$$
$$V_0 = \kappa_1 \cdot V_\kappa$$
$$I ::= V_\kappa$$

$R3_4$: $A ::= \{s22, r21\}$

$$P ::=$$
$$V_\kappa = \kappa \cdot V_0$$
$$V_0 = r21 \cdot s22 \cdot V_0 + \kappa_1 \cdot V_\kappa$$
$$I ::= V_\kappa$$

Note that the parameters of the various V_κ in the patterns are unspecified and that some variable definitions are irregular. We can linearize the irregular transformation rules *for verification purposes.*These are global variables. We define a synchronization law set V_{R3} as follows:

$V_{R3} = \{(\langle s11, r11\rangle, \tau), (\langle s12, r12\rangle, \tau), (\langle s21, r21\rangle, \tau), (\langle s22, r22\rangle, \tau)\}$

We now have all of the elements of a rule system $R3$ defined as follows:

$R3 = \langle [R3_1 \ldots R3_4], V_3, V_{R3}\}\rangle$

$R3$ introduces channels between the producer and the consumer, as intended. Note that the latter two transformation rules of $R3$ require the expansion of the process network by two new dummy processes, and that the synchronization law part of $R3$ replaces the complete synchronization law set.

Chapter 6

Conclusion

6.1 Summary

We showed that process algebra can indeed be used to define both systems containing multiple communicating components, and transformations crossing those components, in ways equivalent to those of the existing LTS-based model transformation method.

We showed that transformation rules involving data parameters and transformation rules which involve changing component configurations can also be defined in a natural way in a process algebra setting, such that symbolic model checking can be used to verify transformations involving forms of infinite behaviour.

Using our lifted and extended model transformation method it is now feasible to model systems and to transform the resulting models with more complex and realistic features. A theoretical basis has been provided onto which successors in this line of research can build a more versatile MDSE based model refinement toolset.

6.2 Future work

A defense of the usefulness of the definitions presented in this thesis would be well served by the implementation of a matching algorithm and by the application of the refinement method to a number of examples. We did not reach that point due to time constraints.

The refinement method presented in this thesis has been designed with the mCRL2 toolset in mind. The mCRL2 tool mCRL22LPS can turn process networks with data into LPS with data, and the mCRL2 tool LPSBisim2PBES can check the bisimilarity of two LPS with data. Unfortunately, LPSBisim2PBES is not fully general with respect to LPS which communicate data variables with infinite domains. Increasing the power of the tool would increase the practical use of our method.

The existing method and our extended lifted method have been proven correct for (rooted) branching bisimulation, but not divergence-preserving branching bisimulation (DPBB). The latter is required for proofs regarding certain kinds of safety and liveness properties. We consciously limit our method to branching bisimulation because the tool LPSBisim2PBES, which is critical in verifying our model transformations, is also limited to branching bisimilarity. As far as we understand this limitation of LPSBisim2PBES is not based on an inherent limitation of the underlying theory, and the tool can be extended to include DPBB. In that case it would be useful to extend our method further to include the latter equivalence as well.

We extended the refinement method to be able to model data, but our extended method still cannot model time or stochastics in an efficient way. An obvious direction for future work would be to explore the consequences of including these two phenomena in our models and transformations in a natural way.

A limitation of the existing method which we have so far adopted is that the property set which we maintain during the refinement process can not itself be subject to transformation. This may be a serious limitation when using the refinement method during the development phase of a software engineering project, because requirements may only become apparent while the development process is already underway, and may even reference concerns which did not exist in earlier, higher-level versions of the model. A way to refine properties, perhaps by including exit variables from the model in the properties as placeholders, would be a useful addition to the refinement method.

A last suggestion for future work is not an extension of the method but a form of support for its users. The application of the refinement method would be very well served by the existence of a library of generally useful transformations which are related to common design patterns. Such a library does not currently exist.

Bibliography

[1] Artemis-ju - aipp. https://artemis-ia.eu/aipp.html.

[2] Artemis-ju - emc2. http://www.artemis-emc2.eu/.

[3] Ecsel-ju. http://www.ecsel-ju.eu/web/index.php.

[4] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R Cordy. A tridimensional approach for studying the formal verification of model transformations. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 921–928. IEEE, 2012.

[5] Jos CM Baeten, Twan Basten, and MA Reniers. *Process algebra: equational theories of communicating processes*, volume 50. Cambridge university press, 2010.

[6] Artur Boronat, Reiko Heckel, and José Meseguer. Rewriting logic semantics and verification of model transformations. In *Fundamental Approaches to Software Engineering*, pages 18–33. Springer, 2009.

[7] Taolue Chen, Bas Ploeger, Jaco Van De Pol, and Tim AC Willemse. Equivalence checking for infinite systems using parameterized boolean equation systems. In *CONCUR 2007–Concurrency Theory*, pages 120–135. Springer, 2007.

[8] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[9] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[10] Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[11] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering*, pages 72–86. Springer, 2007.

[12] E Allen Emerson and Edmund M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer programming*, 2(3):241–266, 1982.

[13] LJP Engelen. From napkin sketches to reliable software. *Faculty of Mathematics and Computer Science, TU/e*, 11, 2012.

[14] LJP Engelen and AJ Wijs. Checking property preservation of refining transformations for model-driven development. Technical Report CS-Report 12-08, Eindhoven University of Technology, 2012.

[15] Miguel Garcia and Ralf Möller. Certification of transformation algorithms in model-driven software development. *Software Engineering*, 105:107–118, 2007.

[16] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck Van Weerdenburg. *The formal specification language mCRL2*. Citeseer, 2007.

[17] Jan Friso Groote, Alban Ponse, and Yaroslav S. Usenko. Linearization in parallel pcrl. *The Journal of Logic and Algebraic Programming*, 48(1):39–70, 2001.

[18] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of model synchronization based on triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 668–682. Springer, 2011.

[19] Jochen M Küster. Definition and validation of model transformations. *Software & Systems Modeling*, 5(3):233–259, 2006.

[20] Frédéric Lang. Exp. open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In *Integrated Formal Methods*, pages 70–88. Springer, 2005.

[21] Levi Lúcio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In *Model Driven Engineering Languages and Systems*, pages 136–150. Springer, 2010.

[22] Sebastiaan Pascal Luttik. Choice quantification in process algebra. 2002.

[23] Radu Mateescu and Anton Wijs. Property-dependent reductions for the modal mu-calculus. In *Model Checking Software*, pages 2–19. Springer, 2011.

[24] Alban Ponse. Computable processes and bisimulation equivalence. *Formal Aspects of Computing*, 8(6):648–678, 1996.

[25] S.M.J. de Putter. On the formal correctness of a model transformation verification technique. Technical Report Master Computer Science Engineering (CSE) 418, Eindhoven University of Technology, 2014.

[26] Lukman Ab Rahim and Jon Whittle. A survey of approaches for verifying model transformations. *Software & Systems Modeling*, 14(2):1003–1028, 2013.

[27] Martin Strecker. Modeling and verifying graph transformations in proof assistants. *Electronic Notes in Theoretical Computer Science*, 203(1):135–148, 2008.

[28] Dániel Varró and András Pataricza. Automated formal verification of model transformations. *CSDUML*, pages 63–78, 2003.

[29] Anton Wijs. Define, verify, refine: Correct composition and transformation of concurrent system semantics. In *Formal Aspects of Component Software*, pages 348–368. Springer, 2014.

[30] Anton Wijs. Confluence detection for transformations of labelled transition systems. *arXiv preprint arXiv:1504.02610*, 2015.

[31] Anton Wijs and Luc Engelen. Refiner: Towards formal verification of model transformations. In *NASA Formal Methods*, pages 258–263. Springer, 2014.

[32] Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schönböck, and Wieland Schwinger. Right or wrong?-verification of model transformations using colored petri nets. In *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM09). Helsinki Business School*, 2009.

[33] Yan Zhang and Yulin Ding. Ctl model update for system modifications. *Journal of Artificial Intelligence Research*, pages 113–155, 2008.