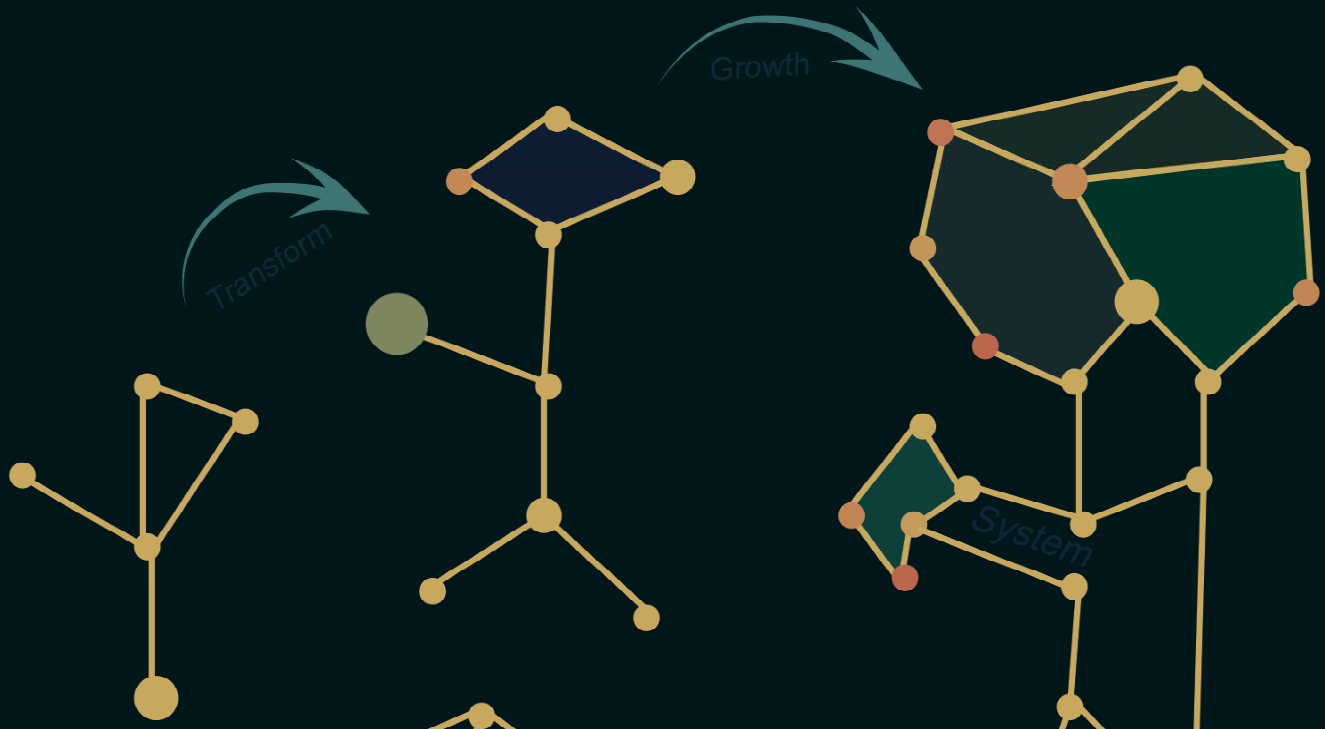
Verification of Concurrent Systems in a Model-Driven Engineering Workflow



Sander de Putter

Verification of Concurrent Systems in a Model-Driven Engineering Workflow

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. F.P.T. Baaijens, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op maandag 28 januari 2019 om 16:00 uur

door

Sander Michaël Jozef de Putter

geboren te Oostburg

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
promotor:	prof.dr. M.G.J. van den Brand
copromotor:	dr.ing. A.J. Wijs
leden:	prof.dr. J.H. Geuvers
	dr.ir. T.A.C. Willemse
	prof.dr. T. Margaria (University of Limerick)
	prof.dr.ir. A. Rensink (University of Twente)
	dr. F. Lang (INRIA)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Verification of Concurrent Systems in a Model-Driven Engineering Workflow

Sander M. J. de Putter

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

Additional members of the reading committee:

prof.dr. J.H. Geuvers	(Eindhoven University of Technology)
dr.ir. T.A.C. Willemse	(Eindhoven University of Technology)
prof.dr. T. Margaria	(University of Limerick)
prof.dr.ir. A. Rensink	(University of Twente)
dr. F. Lang	(INRIA)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).
IPA dissertation series 2018-21.

The work in this thesis has been carried out as part of the Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments (EMC2) project. The EMC2 project has been funded by the AIPP5 programme under grant agreement nº 621429.

A catalogue record is available from the Eindhoven University of Technology Library
ISBN: 978-90-386-4678-7

Printed by: ProefschriftMaken

Cover design: Sander de Putter

Acknowledgements

The seed of this thesis was planted about five years ago when I had my first discussion with my supervisor, dr. Anton Wijs, about verification of model transformations. I ended up doing my Master's graduation project with him on that topic. Thanks to his knowledge, support, and positive attitude, I decided back then that I wanted to learn more from him. Hence, when prof. dr. Mark van den Brand took me on as his PhD student and offered me the opportunity to continue working with Anton as a PhD candidate I accepted immediately. Anton, I enjoyed our many discussions in and outside the office. We sure laughed a lot at those social events. It has been a pleasure working with you.

I want to express my deepest gratitude to my promotor, prof. dr. Mark van den Brand. Mark, you have been a great mentor to me. Your guidance and support have helped me through the ups and downs of PhD-life. Thank you for allowing me to pursue the research interests that have resulted in this thesis.

In the first year of my PhD I took over the organisation of the colloquia of the Software Engineering Technology group from dr. Yanja Dajsuren. Thank you for teaching me about organisation of events and attracting speakers. Furthermore, I would like to thank our secretary, Margje Mommers - Lenders, for her assistance in the organisation of the colloquia. Margje, your skills have been invaluable to the continued success of the colloquia. I also want to express my gratitude to all of the participants of the SET colloquia for their enthusiasm, interests, and talks. In particular, dr. Alexander Serebrenik and dr. Tom Verhoeff have gone above and beyond to offer concrete feedback and introduce more people to the colloquia.

I would like to offer my special thanks to the additional members of the reading committee: prof. dr. Herman Geuvers, prof. dr. Arend Rensink, prof. dr. Tiziana Margaria, dr. Tim Willemse, and dr. Frédéric Lang. I appreciate your kind words and am very grateful for your helpful comments.

I also appreciate the interaction with the members of the EMC^2 project. In particular, I would like to thank: Bastijn Vissers, Tjerk Bijlsma, Reinder Bril, Per Lindgren, and Frank Oppenheimer. Here, I would also like to mention Per's PhD student, Marcus Lindner, who showed me around in Luleå University of Technology.

I also learned a great deal during my time at the PhD-PDEng council of the department of Mathematics and Computer Science. Sarah Gaaf, you were an inspiring chair. Christine van Vredendaal, you were able to keep meetings concise. Jorn van der Pol, you were

always well dressed and to the point. Britt Mathijsen, you were a great treasurer and a good negotiator. I want to thank you and the other members of the PhD-PDEng Council for a great time and the opportunity to grow.

My PhD experience has been all the more enjoyable by my good friends and colleagues at the Software Engineering Technology and Formal System Analysis groups. I have had many memorable moments with my current and past office mates: Yuexu (Celine) Chen, Josh Mengerink, Raquel Álvárez Ramirez, Arash Khabbaz Saberi, Jouke Stoel, and Valcho Dimitrov. Thank you, Mahmoud Telebi, I have learned a great deal by going to the gym together, and moreover, I enjoyed many good conversations and dinners with you. Dan (Dana) Zhang, your cheerfulness was able to brighten everyone's day. Thank you, Ulyana Thikonova, you were always helpful and offered many good tips. Yaping (Luna) Luo, I am grateful for the many trips we made and the board game nights we have had! I also appreciate all the effort Weslley Silva Torres and Thomas Neele put in organising activities together. My thanks also go out to (in no particular order) Önder Babur, Sarmen Keshishzadeh, Priyanka Karkhanis, Felipe Ebert, Ruurd Kuiper, Ramon Schiffelers, Kousar Aslam, Sangeeth Kochanthara, Maricio Verano Merino, Rodin Aarssen, Miguel Botto Tobar, Rob Faessen, Omar Alduhaiby, Bogdan Vasilescu, Alexander Fedotov, Loek Cleophas, Jurgen Vinju, Erik de Vink, Hans Zantema, Julien Schmaltz, Jan Friso Groote, Lou Somers, Ion Barosan, Bas Luttik, Wieger Wesselink, Gerard Zwaan, Maurice Laveaux, Muhammad Osama, Mauricio Verano Merino, Rick Erkens, Serguei Roubtsov, Reinier Post, Erik Scheffers, and other colleagues. Thanks to you all, I have had a wonderful time at the university!

I also would like to thank Vrije Universiteit Amsterdam for their generosity in supplying the computing resources for the experiments performed in this thesis.

I am especially grateful to my family for their unwavering support and love. My thanks go out to my brothers, Robin de Putter, John Ammann de Putter, and Steve Ammann, my sister, Laura de Putter, my parents, Brigitte Buyle and Levien de Putter, my grandparents, Alberic Buyle and Jeanine Schoutteten, and their partners. Additionally, I want to express my gratitude to my girlfriend, Dan Thao Vy , for sharing this journey with me, and her parents. Finally, to all those that I have not mentioned here: thank you!

Sander M. J. de Putter
Eindhoven, December 2018

Table of Contents

List of Acronyms

ACC	Adaptive Cruise Control
CART	Classification and Regression Trees
DPBB	Divergence Preserving Branching Bisimulation
DSL	Domain Specific Language
FIFO	First In First Out
GLMER	Generalized Linear Model with Elasticnet Regularization
HTM	Hardware Transactional Memory
KNN	K-Nearest Neighbours
LDA	Linear Discriminant Analysis
LOWESS	LOcally WEighted Scatterplot Smoothing
LR	Linear Regression
LTS	Labelled Transition System
LVQ	Learning Vector Quantization
MAE	Mean Absolute Error
MDE	Model-Driven Engineering
POR	Partial Order Reduction
QRNN	Quantile Regression Neural Network
RF	Random Forests
SC	Sequentially Consistent
SLCO	Simple Language of Communicating Objects
SOS	Structural Operational Semantics
SVMwLK	Support Vector Machines with Linear Kernel
SVMwRK	Support Vector Machines with Radial Kernel

Chapter 1

Introduction

Concurrent systems form an integral part of today's society. From smartphones, desktops and web systems to the car you drive, and even your coffee machine, concurrent systems can be found everywhere. For instance, a modern car has an Adaptive Cruise Control (ACC) with lane assist controlled by a number of electronic control units that read sensors, process video image, and control the engine, brakes, and steering wheel.

Concurrency in systems has many benefits; e.g., better performance and better distribution of services. However, due to their non-deterministic nature concurrent systems are also more complex, harder to understand, and harder to develop than sequential programs. It is only natural that it is extremely hard to guarantee the correctness of concurrent systems.

Researchers and practitioners have sought to alleviate complexity, increase understandability, and facilitate the early and automated detection of faults. To this end, formal methods and Model-Driven Engineering [185] (MDE) are widely applied [16, 150, 210]. In particular, automated formal methods that integrate well with modern MDE work-flows.

In this thesis we investigate automated formal methods to determine and guarantee correctness of concurrent systems and the integration of formal methods with MDE.

Model-Driven Engineering (MDE) In MDE, models and model transformations are the primary artefacts in the development of a system.

Models are abstract descriptions of real-world systems; they serve as a specification for desired structural and functional constraints of the system. Often models are developed by a domain expert and described in domain specific terms familiar to the expert [210]. A model, as a description of a system, may be used to perform analysis and generate code, tests, and documentation.

Model transformations describe how to obtain one or more output-models from one or more input-models. Amongst other, model transformations are used to: address consistency between models (model synchronisation [88, 223]), refine models (e.g., from platform independent models to platform dependent models [156]), and obtain derived models amenable to analysis [77, 118].

MDE facilitates detection of defects early in the development cycle [104]. Formal semantics of the models allow application of various formal techniques (e.g., verification) or semi-formal inspection by domain experts (e.g., simulation) to find defects. Hence, defects may be repaired early on, thus, preventing costly repairs in later development phases [28].

In this thesis we study the formal analysis of models and model transformations for concurrent systems.

Formal Analysis of Models Over the years, various formal methods have been proposed and further developed to determine the functional correctness of concurrent systems.

In *formal analysis* a model, serving as a formal representation of the concurrent system, is verified against a number of requirements. Requirements are expressed in the form of some formal logic; e.g., in Hoare logic [102] pre- and post-conditions are expressed in predicate logic, and in model checking [16] requirements are expressed in a propositional and temporal modal logic. The formal analysis then determines whether the formal behaviour described by the model satisfies these formal requirements. Thus, formal analysis gives guarantees that a model meets the specified requirements.

With the model serving as a formal specification of the concurrent system, the expectation is that the implementations of the concurrent system will also meet the requirements. This is even more so the case when the implementation is generated from the verified model as is advocated by MDE. Confidence in the satisfaction of the requirements can be strengthened further by generating the implementation using a verified generator as in the work of Zhang et al. [225, 226].

Recall the earlier example of an ACC. When the ACC is active the car should remain at a safe distance from vehicles in front of the car. At the same time the ACC should attempt to drive the car at a desired speed. These requirements may conflict, but the former requirement is safety critical, while the latter is mission critical. Hence, the mission critical requirement must be constrained by the safety critical requirement. Given a (formal) model of the ACC, formal analysis can verify whether the model indeed respects these requirements and their constraints.

Models of Concurrency A field of formal methods specifically focused on concurrent systems is *concurrency theory*. In the early sixties Carl Adam Petri proposed Petri nets [168]; one of the first formalisms for describing concurrent systems. Since then numerous formalisms have been proposed for modelling and reasoning about concurrent systems [79, 190].

To target concurrent systems process calculi and process algebras [54, 103, 149] were designed with parallel composition and synchronisation operators. Formalisation of the semantics of these process calculi and algebras is achieved (amongst others) through *structural operational semantics* (SOS) [98, 170] which specify how the behaviour of the individual processes is interleaved and synchronised. Through the SOS all possible behaviour specified in a concurrent model is described in the form of a *Labelled Transition System* (LTS) [116].[1] Through such formalisations specification of formal requirements over (formal) models is supported.

Moreover, formalisations facilitate the comparison of concurrent systems. This allows one to check whether two models are indeed describing the same concurrent system or to

[1] originally called *named transition system* by Keller [116]

translate a model into a minimal equivalent one. The latter is a frequently used technique to reduce the running-time of analysis algorithms. To compare models with respect to different classes of formal requirements a plethora of pre-orders and equivalence relations have been proposed. An overview of these relations is given by van Glabbeek [205]. In this thesis, we consider *branching bisimulation* and *Divergence-Preserving Branching Bisimulation* (DPBB) [204].

The automatic formal verification of LTSs and specifications written in process calculi and process algebras against formal properties (such as formal requirements) by means of exhaustive exploration is referred to as model checking [16].

Model Checking for Models of Concurrent Systems Much like in MDE, in model checking models are the primary artefact. Model checking has been successfully employed to verify both hardware and software [109, 198].

In *model checking* a model, as a formal representation of the concurrent system, is verified against a number of requirements. The requirements are expressed as formal *properties* of the model expressed in some formal logic (e.g., CTL* [70] and modal μ-calculus [123]).

Given a model and a set of properties a model-checker determines whether the formal behaviour described by the model satisfies these formal properties. The behavioural semantics of the model define how the system may move from one logical state to the next. The model-checker exhaustively traverses the reachable logical states described by the behavioural semantics of the model, also called the *state-space*. A state s is *reachable* iff there is a path from an initial state (e.g., the configuration at start-up) of the system to state s. A property is *refuted* by finding counter-example. A *counter-example* is a structure that can be simulated by the system under scrutiny showing that the property is violated. Usually, such a structure takes the form of a reachable fragment of the state space of the system under scrutiny. When no counter-example is found the property must hold in all states of the model, and thus, the model is guaranteed to satisfy the property.

Although model checking is very promising for the analysis of systems it has one significant problem that impedes its successful application for large concurrent systems: the *state-space explosion problem* [46]. The size of the state-space of a concurrent system is exponential in the number of parallel components. As the run-time complexity of model checking is heavily dependent on size of the state space [35], the model checking of concurrent systems can quickly become infeasible.

Research in model checking has seen a high focus on tackling the state space explosion in the past couple of decades [46]. The results range from modelling guide lines [92] to new model checking algorithms [46]. Techniques such as *symmetry reduction* [44] and *compositional reasoning* [47, 55, 160] approaches take advantage of the symmetric and hierarchical structure of hardware and software systems. Commutativity of concurrently executed transitions is exploited by *partial order reduction* [166]; here one interleaving order is selected as a representation of all possible interleavings of the corresponding concurrent transitions. *Abstract interpretation* [52, 139] allows one to approximate infinite or very large finite transition systems by (smaller) finite ones. To the same end, the state space is represented implicitly in *symbolic model checking* [146] techniques by employing data structures that classify sets of states. Both methods abstract valuations into categorical values avoiding blow-up caused by the numerous values that, for instance, variables may take. The difference between abstract interpretation and symbolic model checking is that abstract interpretation may lose information (i.e., it is an approximative technique) [53]. As abstractly interpreted models are abstract and approximate, the

properties verified on these models are abstract and approximate as well. *Bounded model checking* [22] bounds the search to counter-examples of a given size foregoing completeness. One can attempt to regain completeness by applying *k-induction* [57] or *interpolation* [147], though these approaches may still suffer from infeasibly large search spaces. Yet other works aim to avoid state space explosion by forming contracts of interaction between concurrent components such that they can be verified in (relative) isolation [24, 37, 95].

All these approaches have been applied with some measure of success. None of the approaches is a definitive 'holy grail' of model checking, i.e., none of the approaches works well for all kinds of systems. Furthermore, as the size of concurrent systems is ever increasing, further advancements will remain a popular and necessary topic of research.

Although the state space explosion problem and the verification of models has enjoyed much attention, the verification of other areas of the MDE work flow have not. For the successful application of model checking concurrent models throughout the MDE work flow a number of challenges must be addressed. We will discuss these challenges in the next section.

1.1 Problem statement

The design of a concurrent system starts with the development of models. To verify these models a model checker (e.g., mCRL2 [54] or Cadp [81]) and model checking techniques are chosen and applied. Careful consideration must be given to the properties, the model, and the model checking technique in order to avoid the state space explosion problem.

Once the model has been verified, the model may be implemented by programmers or the implementation may be generated from the model. However, as verification experts often work on a high-level of abstraction to keep the state space small, it may not always be possible to implement the system directly from the model. In this case, the model is refined manually by the designers or automatically via model transformations. These model transformation are specified with mature technologies such as QVT [94], Epsilon [121], ATL [21], and triple graph grammars [186]. When the model has been sufficiently refined the implementation is constructed (automatically or manually).

Despite the increasing maturity of the technologies involved in this work flow, the development of reliable concurrent systems is still met with a number of challenges.

1. **Semantics correctness of model transformations** Despite the maturity of model transformation technologies the verification of model transformations with respect to dynamic semantics has received little attention [173]. All kinds of transformations are in desperate need of thorough verification techniques, especially when they are used in safety critical systems. Transformations such as migration, refactoring, and refinement all end up in the final implementation. If generated implementations and transformed models do not formally adhere to the specification all guarantees are void.

   The closer the model serving as specification is to the actual implementation, the more confidence one has in the correctness of the implementation. If the state space does not increase dramatically due to the transformation the derived models can be re-verified. However, verification of derived models is costly. In particular, for transformations that are reused often, or when the size of the state space does increase too much, re-verification is infeasible [11]. Therefore, it is vital to develop techniques that can verify whether a transformation preserves certain guarantees for all possible input-models.

2. **Congruences for parallel composition** Concurrent systems are specified in a natural way using parallel composition and synchronisation operators. There are many model checking techniques making use of the fact that certain equivalence relations are congruences for parallel composition with synchronisation. The state space explosion can be dampened by minimising the behaviour of the individual parallel processes modulo an appropriate equivalence relation, and possibly its context, before composition of the system.

   Although it is generally assumed that DPBB is a congruence for parallel composition with synchronisation, no proof has been published (e.g., CADP supports compositional reduction of concurrent models modulo DPBB). Since DPBB is the finest in the linear time – branching time spectrum of van Glabbeek [205] that support abstraction, it preserves more properties as other equivalence relations that support abstraction. Hence, it is desirable to apply DPBB as a first state space minimisation step.

3. **Selection of verification approach** There are numerous model checking tools and techniques. To select one for a verification task is a daunting task; especially, since it is often not obvious what technique offers the best (or even reasonable) performance for a given verification task. In short, it is unclear when one technique can be expected to perform better than others.

   Regardless, the literature gives little statistical consideration to how the performance of their model checking techniques may be related to their input. Especially, in the field of explicit-state model checking, many scientists put little effort in the investigation of their model checking techniques beyond a benchmark considering a few models. Although heuristics are developed for model checking techniques, the evaluation of these heuristics often lacks statistical rigour. Furthermore, characteristics of models can be exploited for a number of applications within model checking [163]. Identification of such characteristics must be supported through statistical rigorous findings.

4. **Sequential consistency of concurrent models** A *memory model* specifies the order in which a thread's memory accesses become visible to other threads in a program. A memory model affects both programmability and performance restricting reordering of memory accesses. Strong restrictions reduce the difference between program order and execution order; and therefore increase understandability. Weak restrictions offer more optimisation freedom.

   Programmers expect their code to be executed in the order specified in their program. This model for concurrent programs, called sequential consistency, was formalized by Lamport [131]. Sequential consistency states that all operations of a program are executed in a total order, i.e., atomically and in the order specified by the program. That is, a program is executed in program order and memory accesses are serviced from a single First In First Out (FIFO) queue. A program that follows this model is said to be *Sequentially Consistent* (SC).

   Sequential consistency is arguably the most natural and easy to understand memory model for concurrent systems, but is very restrictive. It is sufficient, however, that a program execution is *indistinguishable* from an SC execution. We call such programs *observably* SC.

   However, most modern compilers and hardware do not guarantee preservation of SC

semantics [29, 142]. Hence, sequential consistency may no longer be guaranteed once an implementation is executed on modern hardware. In fact, interleaving orders that break intended behaviour are a major source of concurrency bugs [140]. To guarantee that the results of model checkers are still applicable in the obtained implementation the SC semantics of concurrent models must be preserved in generated (or implemented) program code.

1.2 Research Questions

For each of the problems stated in Section 1.1 we formulate a corresponding research question. Additionally, we dedicate a research question on the application of our results in MDE.

Current model transformation verification techniques are mainly focused on verifying preservation of well-formedness or static semantics [210]. Techniques that do support verification of dynamic semantics do not have intrinsic support for concurrency. To gain support for verification of transformations of concurrent models we formulate the following research question:

> **RQ 1:** *How can we verify preservation of dynamic semantics of model transformations of concurrent models?*

One of the greatest facilitators for the verification of concurrent systems are congruences for parallel composition operators that also perform synchronisations. Although it has been shown for many equivalence relations that they are congruences for such parallel composition operator, no such results exist for DPBB. To eliminate any doubt we investigate the following research question:

> **RQ 2:** *Is DPBB a congruence for parallel composition with synchronisations?*

The selection of verification techniques amongst the numerous alternatives is often a daunting task. As a first step towards a heuristic for assisting in choosing a verification technique, we investigate the memory performance of compositional aggregation and compare it with a monolithic minimisation as base line. Compositional aggregation has shown to perform better (in the size of the largest state space in memory at one time) than classical monolithic composition in a number of cases. However, there are also cases in which compositional aggregation performs much worse. It is unclear when one should apply compositional aggregation in favour of other techniques and how it is affected by aggregation order, action hiding and the scale of the model. With this goal in mind we get the following research question:

> **RQ 3:** *When can compositional aggregation be expected to be more (memory) efficient than monolithic minimisation?*

Due to modern compiler and hardware optimisations non-SC behaviour may be observed during execution of a program. By appropriately using synchronisation mechanisms such as semaphores and atomic instructions one can ensure that a program is observably SC [188]. However, automatic algorithms for restoring observable sequential consistency are often quite pessimistic since not all dependencies can be determined a priori through static dependency analysis [13, 181].

In this thesis, we investigate how sequential consistency can be guarantee during the execution of code generated from concurrent models:

> **RQ 4:** *How can sequential consistency be preserved (up to observation) from model to execution of generated code?*

The results obtained from the previous research questions are likely to apply to some underlying mathematical model. Our last research question investigates how to integrate these results in an MDE work flow such that they become accessible for developers.

> **RQ 5:** *How can our results be applied in an MDE context?*

1.3 Outline and Origin of Chapters

In this section, we present a brief overview of each chapter, we indicate the research questions addressed and point out the chapter's origin.

Chapter 2 This chapter briefly discusses the formal notions used throughout this thesis. The behaviour of a processes of a concurrent system is described by a Labelled Transition System (LTS). This thesis focusses on branching bisimulation and DPPB as equivalences for LTSs. To specify a concurrent system, an LTS network is used. An LTS network contains a vector of LTSs and a set of synchronisation laws. The (global) semantics of an LTS network are described by its system LTS. A system LTS is constructed through the parallel composition of the LTSs in the network's vector of LTSs subject to the set of synchronisation laws. Furthermore, the notion of a congruence for LTS networks is introduced. Finally, admissibility of an LTS network is presented as branching bisimulation and DPBB are only congruences for LTS networks that are admissible.

Chapter 3 This chapter addresses research question RQ 1.

We propose a formal verification technique to determine that formalisations of transformations in the form of rule systems are guaranteed to preserve functional properties, regardless of the models they are applied on. Rule systems consist of a set of transformation rules over LTSs, a set of synchronisation laws that are expected in the input system, and a set of synchronisation laws that will be introduced. A transformation rule specifies an LTS pattern to match and its replacement.

When an abstracted state space is constructed from the matching and replacing LTS patterns, preservation of a given functional property is verified by comparing the two state spaces modulo branching bisimulation.

This chapter is taken from

> [60] de Putter, S., and Wijs, A. A formal verification technique for behavioural model-to-model transformations. *Formal Aspects of Computing* (Oct 2017)

a special issue extension of

> [59] de Putter, S., and Wijs, A. Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique. In *FASE 2016* (2016), vol. 9633 of *LNCS*, Springer, pp. 383–400

Chapter 4 In this chapter, research question RQ 2 is addressed.

DPBB is the finest equivalence relation in the linear time - branching time spectrum of van Glabbeek. Therefore, it is desirable to use in a first state space minimisation step. This chapter finally proves that DPBB is indeed a congruence for parallel composition with synchronisation between components.

We discuss commutativity and associativity of parallel composition with synchronisation in the context of DPBB. Moreover, we show how to decompose an existing specification of a concurrent system into sub-components that is consistent with the original specification.

This chapter is taken from

> [58] de Putter, S., Lang, F., and Wijs, A. Compositional Model Checking is Lively - Extended Version. *Science of Computer Programming* (2019). Special Issue on FACS. *Manuscript under review*

a special issue extension of

> [64] de Putter, S., and Wijs, A. J. Compositional Model Checking Is Lively. In *FACS* (2017), vol. 10487 of *LNCS*, Springer, pp. 117–136

which received the FACS 2017 *Best Student Paper Award*.

Chapter 5 In this chapter, we address research question RQ 3.

This chapter takes a first step towards statistically supported selection of model checking approaches based on model characteristics. Following the quantitative experimental approach, this chapters presents a descriptive analysis that compares compositional aggregation with monolithic minimisation. Furthermore, we apply machine learning to create predictors for the efficiency of two heuristics compared to monolithic minimisation.

This chapter is an extension of

> [61] de Putter, S., and Wijs, A. To Compose, or Not to Compose, That Is the Question: An Analysis of Compositional State Space Generation. In *FM* (2018), Springer International Publishing, pp. 485–504

Chapter 6 In this chapter, we address research question RQ 4.

Due to modern compiler and hardware optimisations non-SC behaviour may be observed during execution of a program. Shasha and Snir [188] proposed an algorithm that applies a minimal number of locks and inserts a minimal number of delays. In this chapter, we encode part of this algorithm as a model checking technique , and we use the concurrent model as specification for program order and atomicity.

The output of the algorithm is used to add annotations to the concurrent model. These annotations are then used by the code generator of Zhang et al. [225, 226] to generate efficient code that employs a minimal set of blocking statements.

This chapter has been submitted to ESOP 2019

> [62] de Putter, S., and Wijs, A. Model Driven Avoidance of Atomicity Violations under Relaxed-Memory Models. In *ESOP* (2019). *Submitted*

Chapter 7 In this chapter, we address research question RQ 5. The chapter presents the Simple Language of Communicating Objects (SLCO) [71, 224], a Domain Specific Language (DSL) for modelling concurrent state machines, and the SLCO framework built around SLCO. The framework supports verification of SLCO models via MCRL2 [54]. Furthermore, the framework offers a verified Java code generator [224, 225]. Within the work of this thesis the SLCO framework was extended with a prototype transformation verification method powered by the theory of Chapter 3. In Addition, the code generator was extended to support minimal insertion of synchronisation mechanisms that preserves sequential consistency with respect to the source SLCO model by employing the algorithm proposed in Chapter 6. Finally, verification of SLCO models may proceed compositionally as discussed in Chapter 4.

This chapter is an extension of

> [63] DE PUTTER, S., WIJS, A., AND ZHANG, D. The SLCO Framework for Verified, Model-Driven Construction of Component Software. In *FACS* (2018), LNCS, Springer, pp. 288–296

Chapter 8 This chapter concludes the thesis. It revisits the research questions and contributions. Finally, directions for future research are proposed.

1.4 Suggested Method of Reading

Before reading Chapters 3 to 5 one should familiarise oneself with the notions introduced in Chapter 2. Apart from those notions, these chapters are self-contained and can be read independently. Chapters 6 and 7 are fully self-contained and do not require any prior knowledge contained in other chapters.

Chapter 3 discusses the verification of transformations. Chapter 4 presents a proof showing that DPBB is a congruence for parallel composition with synchronisations. Chapter 5 deals with compositional aggregation and when it is expected to out-perform monolithic minimisation. Chapter 6 presents an algorithm which can be exploited to generate guaranteed observably SC implementations of concurrent models. Chapter 7 explains how the material presented in the previous chapters is applied in an MDE framework. Finally, Chapter 8 presents the conclusions of this thesis and elaborates on further directions of future work.

Chapter 2

Preliminaries

In this chapter we briefly introduce the notions that are used throughout this thesis. These notations concern the semantics of systems.

A Labelled Transitions System (LTS) describes the behaviour of a process or system. The behaviour of a concurrent system is described by a network of LTSs*, or* LTS network *for short. From an LTS network, a* system LTS *can be derived describing the global behaviour of the network. An LTS or system LTS may be minimised modulo an appropriate equivalence relation.*

To compare or minimise the behaviour of these LTSs and system LTSs an equivalence relation is used. An equivalence relation between two LTSs relates states that have equivalent behaviour. The minimisation of an LTS computes, for a given LTS, a minimum equivalent LTS. In this chapter we introduce the equivalence relations named branching bisimulation *and* divergence preserving branching bisimulation.

2.1 Vectors

A vector $\bar{v}$ of size n contains n elements indexed from 1 to n. We write $1..n$ for the set of integers ranging from 1 to n. For all $i \in 1..n$, $\bar{v}_i$ represents the i^{th} element of $\bar{v}$. The *concatenation* of two vectors $\bar{v}_1$ and $\bar{v}_2$ is denoted by $\bar{v}_1 \parallel \bar{v}_2$. Moreover, x^n is the vector consisting of n elements x.

Consider a set of indices $I \subseteq 1..n$. The indices with the i^{th} rank ($i \in 1..|I|$) among the indices in I is denoted I_i. The *projection* of a vector $\bar{v}$ on to I is defined as the vector $\bar{v}^I = \langle \bar{v}_{I_1}, \ldots, \bar{v}_{I_{|I|}} \rangle$ of length $|I|$. Furthermore, given a vector v, and some element x, the vector with all elements at indices in I *substituted* by x is defined as $v[I \mapsto x] = \langle e_1, \ldots, e_n \rangle$ with $\forall i \in I.\ e_i = x$ and $\forall i \in 1..n \setminus I.\ e_i = v_i$.

2.2 LTS and LTS equivalences

Labelled Transition System. The semantics of a process, or a composition of several processes, can be formally expressed by an LTS as presented in Definition 2.2.1.

Definition 2.2.1 (Labelled Transition System)**.** *An* LTS $\mathcal{G}$ *is a tuple* $(\mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G})$*, with*

- $\mathcal{S}_\mathcal{G}$ *a finite set of states;*
- $\mathcal{A}_\mathcal{G}$ *a set of action labels;*
- $\mathcal{T}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \times \mathcal{S}_\mathcal{G}$ *a transition relation;*
- $\mathcal{I}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G}$ *a (non-empty) set of initial states.*

Action labels in $\mathcal{A}_\mathcal{G}$ are denoted by a, b, c, etc. Additionally, there is a special action label τ that represents internal, or hidden, system steps. A transition $(s, a, s') \in \mathcal{T}_\mathcal{G}$, or $s \xrightarrow{a}_\mathcal{G} s'$ for short, denotes that LTS $\mathcal{G}$ can move from state s to state s' by performing the a-action. The transitive reflexive closure of $\xrightarrow{a}_\mathcal{G}$ is denoted as $\xrightarrow{a}{}^*_\mathcal{G}$, and the transitive closure is denoted as $\xrightarrow{a}{}^+_\mathcal{G}$.

Equivalence of LTSs LTSs equivalence relations allow the comparison and minimisation of LTSs. Two prominent equivalence relations are *branching bisimulation* [207] and *Divergence Preserving Branching Bisimulation* (DPBB) (also known as *branching bisimulation with explicit divergence* [204, 207]). Both equivalence relations support abstraction from actions and are sensitive to internal actions and the branching structure of an LTS. In addition, DPBB is sensitive to cycles of τ-transitions, i.e., infinite internal behaviour. We require abstraction from actions for the verification of abstraction and refinement transformations (Chapter 3) such that input and output models can be compared on the same abstraction level.

A branching bisimulation relation is defined as follows.

Definition 2.2.2 (Branching bisimulation)**.** *A binary relation B between two LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a* branching bisimulation *iff $s\ B\ t$ implies*

(B1) if $s \xrightarrow{a}_{\mathcal{G}_1} s'$ then

(a) either $a = \tau$ with $s'\ B\ t$;

(b) or $t \xrightarrow{\tau}{}^*_{\mathcal{G}_2} \hat{t} \xrightarrow{a}_{\mathcal{G}_2} t'$ *with* $s\ B\ \hat{t}$ *and* $s'\ B\ t'$.

(B2) the symmetric case: if $t \xrightarrow{a}_{\mathcal{G}_2} t'$ *then*

(a) either $a = \tau$ *with* $s\ B\ t'$;

(b) or $s \xrightarrow{\tau}{}^*_{\mathcal{G}_1} \hat{s} \xrightarrow{a}_{\mathcal{G}_1} s'$ *with* $\hat{s}\ B\ t$ *and* $s'\ B\ t'$.

Two states $s \in \mathcal{S}_{\mathcal{G}_1}$ and $t \in \mathcal{S}_{\mathcal{G}_2}$ are *branching bisimilar*, denoted $s \underline{\leftrightarrow}_b t$, iff there is a branching bisimulation relation B such that $s\ B\ t$. Two sets of states $S_1 \subseteq \mathcal{S}_{\mathcal{G}_1}$ and $S_2 \subseteq \mathcal{S}_{\mathcal{G}_2}$ are called branching bisimilar, denoted $S_1 \leftrightarrow_b S_2$, iff there is a branching bisimulation relation B such that $\forall s_1 \in S_1.\exists s_2 \in S_2.s_1\ B\ s_2$ and vice versa. We say that two LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ are branching bisimilar, denoted $\mathcal{G}_1 \underline{\leftrightarrow}_b \mathcal{G}_2$, iff there is a branching bisimulation relation B such that $\mathcal{I}_{\mathcal{G}_1}\ B\ \mathcal{I}_{\mathcal{G}_2}$. Moreover, two LTSs may share states, i.e., it is possible that $s \in \mathcal{S}_{\mathcal{G}_1} \cap \mathcal{S}_{\mathcal{G}_2}$.

A DPBB relation is a branching bisimulation relation with two extra conditions as presented in Definition 2.2.3. To simplify proofs define DPBB with the weakest divergence condition (D_4) presented in [204]. This definition is equivalent to the standard definition of DPBB [204]. The smallest infinite ordinal is denoted by ω.

Definition 2.2.3 (Divergence-Preserving Branching Bisimulation)**.** *A binary relation B between two LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a* divergence-preserving branching bisimulation *iff B is a branching bisimulation and for all $s \in \mathcal{S}_{\mathcal{G}_1}$ and $t \in \mathcal{S}_{\mathcal{G}_2}$, $s\ B\ t$ implies:*

(D1) if there is an infinite sequence of states $(s^k)_{k\in\omega}$ such that $s = s^0$, $s^k \xrightarrow{\tau}_{\mathcal{G}_1} s^{k+1}$ and $s^k\ B\ t$ for all $k \in \omega$, then there exists a state t' such that $t \xrightarrow{\tau}{}^+_{\mathcal{G}_2} t'$ and $s^k\ B\ t'$ for some $k \in \omega$.

(D2) the symmetric case: if there is an infinite sequence of states $(t^k)_{k\in\omega}$ such that $t = t^0$, $t^k \xrightarrow{\tau}_{\mathcal{G}_1} t^{k+1}$ and $s\ B\ t^k$ for all $k \in \omega$, then there exists a state s' such that $s \xrightarrow{\tau}{}^+_{\mathcal{G}_1} s'$ and $s'\ B^k$ for some $k \in \omega$.

Two states $s \in \mathcal{S}_{\mathcal{G}_1}$ and $t \in \mathcal{S}_{\mathcal{G}_2}$ are *divergence-preserving branching bisimilar*, denoted by $s \underline{\leftrightarrow}^{\Delta}_b t$, iff there is a DPBB relation B such that $s\ B\ t$. Two sets of states $S_1 \subseteq \mathcal{S}_{\mathcal{G}_1}$ and $S_2 \subseteq \mathcal{S}_{\mathcal{G}_2}$ are divergence-preserving branching bisimilar, denoted $S_1 \underline{\leftrightarrow}^{\Delta}_b S_2$, iff there is a DPBB relation B such that $\forall s_1 \in S_1.\exists s_2 \in S_2.s_1\ B\ s_2$ and vice versa. We say that two LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ are divergence-preserving branching bisimilar, denoted by $\mathcal{G}_1 \underline{\leftrightarrow}^{\Delta}_b \mathcal{G}_2$, iff there is a DPBB relation B such that $\forall s_1 \in \mathcal{I}_{\mathcal{G}_1}.\exists s_2 \in \mathcal{I}_{\mathcal{G}_2}.\ s_1\ B\ s_2$ and vice versa.

2.3 Concurrent LTSs

LTS Network An LTS network [133] (Definition 2.3.1) describes a system consisting of a finite number of concurrent process LTSs and a set of synchronisation laws which define the possible interaction between the processes. The explicit behaviour of an LTS network is defined by its *system LTS* (Definition 2.3.3).

Definition 2.3.1 (LTS network)**.** *An LTS network $\mathcal{N}$ of size n is a pair $(\Pi, \mathcal{V})$, where*

- Π *is a vector of n concurrent LTSs. For each $i \in 1..n$, we write $\Pi_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i)$ and $s \xrightarrow{a}_i s'$ as shorthand for $s \xrightarrow{a}_{\Pi_i} s'$.*

- $\mathcal{V}$ *is a finite set of synchronisation laws. A* synchronisation law *is a tuple* $(\bar{v}, a)$*, where* $\bar{v}$ *is a vector of size* n*, called the* synchronisation vector*, describing synchronising action labels, and* a *is an action label representing the result of successful synchronisation. We have* $\forall i \in 1..n.\ \bar{v}_i \in \mathcal{A}_i \cup \{\bullet\}$*, where* $\bullet$ *is a special symbol denoting that* Π_i *performs no action.*

Given a set of synchronisation laws $\mathcal{V}$, the *set of result actions* is defined as $\mathcal{A}_\mathcal{V} = \{a \mid (\bar{v}, a) \in \mathcal{V}\}$. Furthermore, the set of indices of processes participating in a synchronisation law $(\bar{v}, a)$ is defined as $Ac(\bar{v}) = \{i \mid i \in 1..n \wedge \bar{v}_i \neq \bullet\}$; e.g., $Ac(\langle c, b, \bullet\rangle) = \{1, 2\}$. Finally, the set of all sets of active indices is defined by $Ac(\mathcal{V}) = \{Ac(\bar{v}) \mid \bar{v} \in \mathcal{V}\}$.

The LTS network model subsumes most hiding, renaming, cutting, and parallel composition operators present in process algebras, but also more expressive operators such as synchronisations of m among n processes [136]. For instance, hiding can be applied by replacing the a component in a law by τ. A transition of a process LTS is *cut* if it is blocked with respect to the behaviour of the whole system (system LTS), i.e., there is no synchronisation law involving the transition's action label at the position of the process LTS.

An LTS network is called *admissible* if the synchronisation laws of the network do not synchronise, rename, or cut τ-transitions [133] as defined in Definition 2.3.2. The intuition behind this is that internal, i.e., hidden, behaviour should not be restricted by any operation. Techniques such as partial model checking and compositional construction rely on LTS networks being admissible [81].

Definition 2.3.2 (LTS network Admissibility)**.** *An LTS network* $\mathcal{N} = (\Pi, \mathcal{V})$ *of length* n *is called admissible iff the following properties hold:*

1. $\forall (\bar{v}, a) \in \mathcal{V}, i \in 1..n.\ \bar{v}_i = \tau \implies \neg\exists j \neq i.\ \bar{v}_j \neq \bullet$; *(no synchronisation of* τ*'s)*
2. $\forall (\bar{v}, a) \in \mathcal{V}, i \in 1..n.\ \bar{v}_i = \tau \implies a = \tau$; *(no renaming of* τ*'s)*
3. $\forall i \in 1..n.\ \tau \in \mathcal{A}_i \implies \exists (\bar{v}, a) \in \mathcal{V}.\ \bar{v}_i = \tau$. *(no cutting of* τ*'s)*

The System LTS of an LTS network The explicit behaviour of an LTS network $\mathcal{N}$ is defined by its *system LTS* $\mathcal{G}_\mathcal{N}$ which is obtained by combining the processes in Π according to the synchronisation laws in $\mathcal{V}$ as specified by Definition 2.3.3. The LTS network model subsumes most hiding, renaming, cutting, and parallel composition operators present in process algebras. For instance, hiding can be applied by replacing the a component in a law by τ.

Definition 2.3.3 (System LTS)**.** *Given an LTS network* $\mathcal{N} = (\Pi, \mathcal{V})$*, its* system LTS *is defined by* $\mathcal{G}_\mathcal{N} = (\mathcal{S}_\mathcal{N}, \mathcal{A}_\mathcal{N}, \mathcal{T}_\mathcal{N}, \mathcal{I}_\mathcal{N})$*, with*

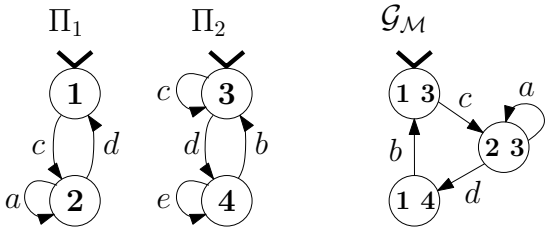
- $\mathcal{S}_\mathcal{N} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$;
- $\mathcal{I}_\mathcal{N} = \{\langle s_1, \ldots, s_n\rangle \mid s_i \in \mathcal{I}_i\}$;
- $\mathcal{T}_\mathcal{N}$ *and* $\mathcal{S}_\mathcal{N}$ *are the smallest relation and set, respectively, satisfying* $\mathcal{I}_\mathcal{N} \subseteq \mathcal{S}_\mathcal{N}$ *and for all* $\bar{s} \in \mathcal{S}_\mathcal{N}$*,* $a \in \mathcal{A}_\mathcal{V}$*, we have* $\bar{s} \xrightarrow{a}_\mathcal{N} \bar{s}'$ *and* $\bar{s}' \in \mathcal{S}_\mathcal{N}$ *iff there exists* $(\bar{v}, a) \in \mathcal{V}$ *such that for all* $i \in 1..n$*:*

$$\begin{cases} \bar{s}_i = \bar{s}'_i & \text{if } \bar{v}_i = \bullet \\ \bar{s}_i \xrightarrow{\bar{v}_i}_{\Pi_i} \bar{s}'_i & \text{otherwise} \end{cases}$$

- $\mathcal{A}_{\mathcal{N}} = \{a \mid \exists \bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{N}}.\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'\}$.

Whenever we want to make explicit that a transition $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ is enabled by a synchronisation law $(\bar{v}, a)$, we write $\bar{s} \xrightarrow{\bar{v},a}_{\mathcal{N}} \bar{s}'$. We refer to $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ and $\bar{s} \xrightarrow{\bar{v},a}_{\mathcal{N}} \bar{s}'$ transitions as *global transitions* and we refer to transitions $\bar{s} \xrightarrow{\bar{v}_i}_i \bar{s}'$ $(i \in Ac(\bar{v}))$ as the *(process-)local transitions*. If it is clear from the context whether a transition is global or local, then "global" or "local" is omitted.

In Figure 2.1, an example of an LTS network $\mathcal{N} = (\langle \Pi_1, \Pi_2 \rangle, \mathcal{V})$ with four synchronisation laws is shown on the left, and the corresponding system LTS $\mathcal{G}_{\mathcal{N}}$ is shown on the right. Initial states are indicated with an incoming arrow. The states of the system LTS $\mathcal{G}_{\mathcal{N}}$ are constructed by combining the states of Π_1 and Π_2. In this example, we have $\langle 1,3\rangle, \langle 1,4\rangle, \langle 2,3\rangle \in \mathcal{S}_{\mathcal{N}}$, of which $\langle 1,3\rangle$ is the single initial state of $\mathcal{G}_{\mathcal{N}}$.



$$\mathcal{V} = \left\{ \begin{array}{l} (\langle c, c\rangle, c), \\ (\langle d, d\rangle, d), \\ (\langle a, \bullet\rangle, a), \\ (\langle \bullet, b\rangle, b), \\ (\langle a, e\rangle, f) \end{array} \right\}$$

Figure 2.1: An LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ (left) and its system LTS $\mathcal{G}_{\mathcal{N}}$ (right)

The transitions of the system LTS in Figure 2.1 are constructed by combining the transitions of Π_1 and Π_2 according to the set of synchronisation laws $\mathcal{V}$. Law $(\langle c, c\rangle, c)$ specifies that the process LTSs can synchronise on their c-transitions, resulting in c-transitions in the system LTS. Similarly, the process LTSs can synchronise on their d-transitions, resulting in a d-transition in $\mathcal{G}_{\mathcal{N}}$. Furthermore, law $(\langle a, \bullet\rangle, a)$ specifies that process Π_1 can perform an a-transition independently resulting in an a-transition in $\mathcal{G}_{\mathcal{N}}$. Likewise, law $(\langle \bullet, b\rangle, b)$ specifies that the b-transition can be fired independently by process Π_2. When the network is in state $\langle 1,4\rangle$ and Π_2 performs the b-action, Π_1 remains in state $\langle 1\rangle$ because Π_1 does not participate in law $(\langle \bullet, b\rangle, b)$. The last law states that a- and e-transitions can synchronise, resulting in f-transitions, however, in this example the a- and e-transitions in Π_1 and Π_2 are never able to synchronise since state $\langle 2,4\rangle$ is unreachable.

Congruence for LTS networks Replacement and minimisation of processes in an LTS network (such as in compositional aggregation) relies on the applied equivalence relation being a congruence for LTS networks. Intuitively, if an equivalence relation R is a congruence for LTS networks, any process Π_i $(i \in 1..n)$ in some network $\mathcal{N} = (\Pi, \mathcal{V})$ may be replaced by an equivalent (modulo R) process, e.g., R) process, e.g., a minimisation of Π_i.

Definition 2.3.4 (Congruence for LTS networks)**.** *An LTS equivalence R is a* congruence for LTS networks *iff for all LTS networks $(\Pi, \mathcal{V})$ of some size n, and all vectors of LTSs Ψ (also of size n) it holds that*

$$(\forall i \in 1..n.\ \Pi_i\ R\ \Psi_i) \implies \mathcal{G}_{(\Pi,\mathcal{V})}\ R\ \mathcal{G}_{(\Psi,\mathcal{V})}$$

Branching bisimulation, DPBB, observational equivalence, safety equivalence and weak trace equivalence, are congruences for *admissible* LTS networks [58, 81] (see also Chapter 4).

Chapter 3

Transformation Verification

In Model Driven Engineering, models and model transformations are the primary artefacts when developing a software system. In such a workflow, model transformations are used to incrementally transform initial abstract models into concrete models containing all relevant system details. Over the years, various formal methods have been proposed and further developed to determine the functional correctness of models of concurrent systems. However, the formal verification of model transformations has so far not received as much attention.

In this chapter, we propose a formal verification technique to determine that formalisations of such transformations in the form of rule systems are guaranteed to preserve functional properties, regardless of the models they are applied to. Given n transformation rules in the rule system, we show that only up to n individual checks are required to determine preservation of a property, whereas previously, up to $2^n - 1$ checks were required. Furthermore, a full correctness proof for the technique is presented, based on a formal proof conducted with the Coq *proof assistant.*

The technique was implemented in Refiner *version 2. We report on two sets of conducted experiments. In the first set, we compared traditional model checking with transformation verification, and in the second set, we compared* Refiner *version2 with the previous version of* Refiner.

This chapter is taken from

[60] de Putter, S., and Wijs, A. A formal verification technique for behavioural model-to-model transformations. *Formal Aspects of Computing* (Oct 2017)

a special issue extension of

[59] de Putter, S., and Wijs, A. Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique. In *FASE 2016* (2016), vol. 9633 of *LNCS*, Springer, pp. 383–400

3.1 Introduction

It is a well-known fact that concurrent systems are very hard to develop correctly. In order to support the development process, over the years, a whole range of formal methods have been constructed to determine the functional correctness of system models [34]. Over time, these techniques have greatly improved, but the analysis of complex models is still time-consuming, and often beyond what is currently possible.

To get a stronger grip on the development process, Model-Driven Engineering has been proposed [119]. In this approach, models are constructed iteratively, by defining *model transformations* that can be viewed as functions applicable to models: they are applied to models, producing new models. Using such transformations, an abstract initial model can be gradually transformed into a very detailed model describing all aspects of the system. If one can determine that the transformations are correct, then it is guaranteed that a correct initial model will be transformed into a correct final model.

Most model transformation verification techniques are focussed on determining that a given transformation applied to a given model produces a correct new model, but in order to show that a transformation is correct in general, one would have to determine this for *all possible input models*. Two survey papers [7, 173] identify some techniques that can do this, but these techniques are often informal or require high effort (e.g., by using a theorem prover).

This work is an extension of [59], where we formally proved the correctness of such a formal transformation verification technique proposed in [214, 217] and implemented in the tool Refiner [218]. It is applicable to models with a semantics that can be captured by Labelled Transition Systems (LTSs). Transformations are formally defined as *LTS transformation rules*. Correctness of transformations is interpreted as the *preservation of properties*. Given a property φ written in a fragment of the μ-calculus [144], and a system of transformation rules Σ, Refiner checks whether Σ preserves φ for all possible inputs. This is done by first hiding all behaviour irrelevant for φ [114, 144] and then checking whether the rules replace parts of the input LTSs by new parts that are *branching bisimilar* to the old ones. Branching bisimilarity preserves safety properties and a subset of liveness properties [207]. Furthermore, for systems in which Kooman's fair abstraction rule [122] holds, branching bisimulation also preserves liveness property involving inevitable reachability [207].

Figure 3.1 provides an overview of the transformation verification workflow in Refiner. Given as input is a rule system consisting of n LTS transformation rules, where each rule r_i consists of a left pattern L_i, describing which component behaviour is subject to transformation, and a right pattern R_i, defining the behaviour produced by a transformation of the corresponding left pattern behaviour. If such a rule system were to be applied to an input model, Refiner would identify the possible matches of the left patterns of the rules on the behaviour of the components in the model, and subsequently, apply the transformation rules to those matches, thereby replacing the existing behaviour with copies of the corresponding right patterns.

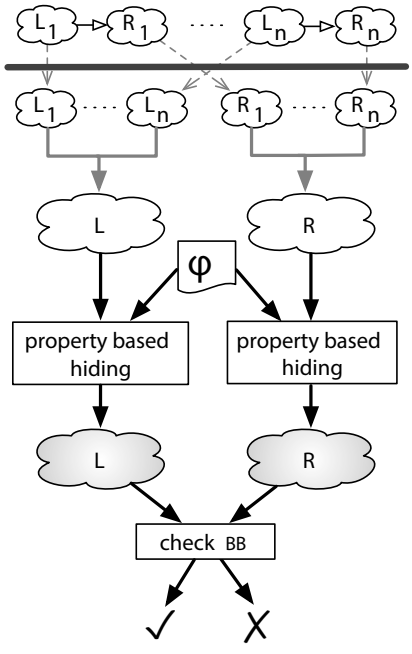


Figure 3.1: LTS transformation verification with Refiner

In order to verify that a rule system will preserve a property φ for any model it is applied to, Refiner combines the left patterns on the one hand, and the right patterns on the other hand. Then Refiner produces the state spaces of both these combinations, as these can be interpreted as models themselves. In practice, Refiner actually checks whether patterns are dependent on each other, in the sense that their behaviour needs to synchronise at some point, and groups the rules together into sets of dependent rules. In this example, there is only one such group.

Next, *property-based hiding* [144] is performed, given a property φ to check. Finally, the resulting abstract state spaces are compared using a branching bisimulation checking algorithm. Only if the combinations of both the left and the right patterns are branching bisimilar will the outcome of this check be positive: indicating that the property φ is preserved in any application of the rule system. If the patterns are not branching bisimilar the φ is not preserved for all applications of the rule system, however, φ may still be preserved for some applications. When no property is considered, the technique checks for full semantics preservation, i.e., it does not apply property-based hiding. This is useful, for instance, when refactoring models.

The technique has been successfully applied to reason very efficiently about model transformations; speed-ups of five orders of magnitude have been measured with respect to traditional model checking of the models produced by a transformation [214]. However, as the technique is theoretically very involved, its absolute correctness, i.e., whether it returns **true** iff a given rule system is property preserving for all possible input models, has been an open question since it was constructed. In [59] we first addressed the correctness of the transformation verification technique. After finding and fixing two issues the verification technique was proven correct.

Contributions This chapter addresses the formal correctness of the transformation verification technique from [214].

First, we have verified transformation rules that distinguish between glue-states that allow incoming and/or outgoing transitions entering or leaving the LTS patterns, respectively. By doing so, the verification technique is able to handle more cases.

Moreover, we present a proof that shows that the required number of bisimulation checks when verifying an LTS transformation rule system can be reduced from $2^n - 1$ per set of dependent transformation rules (where n is the upper bound of the number of rules in such a set) to only one per set of dependent rules. This proof is presented in greater detail than the one given previously [59] and is based on a formal proof conducted with the Coq proof assistant [1] version 8.6 (December 2016). The Coq formalisation is available online. [2]

Structure of the chapter Related work is discussed in Section 3.2. Section 3.3 presents the notions for and analysis of the application of a rule system consisting of only a single transformation rule. A correctness proof is presented. This section can be viewed as a first step towards discussing the complete technique, applicable to rule systems consisting of multiple rules. Next, in Section 3.4, the complete technique is presented; the discussion is continued by considering networks of concurrent process LTSs, and systems of transformation rules. Again, we give a proof of correctness.

[1] http://coq.inria.fr

[2] http://www.win.tue.nl/mdse/property_preservation/FAC2017_LTS_Network_transformation_verification.zip

After that, we present experimental results in Section 3.5, by which we demonstrate the effectiveness of the analysis technique, compared to, more traditional, model checking the models again once they have been altered by a model transformation. Finally, section 3.6 contains our conclusions and pointers to future work.

3.2 Related Work

Papers on incremental model checking (IMC) propose how to reuse model checking results of safety properties for a given input model after it has been altered [191, 195]. We also consider verifying models that are subject to changes. However, we focus on analysing transformation specifications, i.e., the changes themselves, allowing us to determine whether a change always preserves correctness, independent of the input model. Furthermore, our technique can also check the preservation of (a subset of) liveness properties.

In the context of *Dynamic graph algorithms* [72], reachability is an unbounded problem [174, 191], i.e., it cannot be determined solely based on the changes. Thanks to our criteria, this is not an issue in our context.

In [184], an incremental algorithm is presented for updating bisimulation relations based on changes applied to a graph. Their goal is to efficiently maintain a bisimulation, whereas our goal is to assess whether bisimulations are guaranteed to remain after a transformation has been applied without considering the whole relation. As is the case for the IMC techniques, this algorithm works only for a given input graph, while we aim to prove correctness of the transformation specification itself regardless of the input.

In *refinement checking* [1, 127], supported by tools such as Rodin [4], Fdr3[3] and Csp-Casl-Prover [112], it is usually checked that one model refines another. This is very similar to our approach, but refinements are defined in terms of what the new model will be, as opposed to how the new model can be obtained from the old one, i.e., model transformations are not represented as artefacts independent of the models they can be applied to. This makes the technique not directly suitable to investigate the feasibility to verify definitions of model transformations, as opposed to the models they produce.

The Bart tool[4] allows automatically refining B components to $B0$ implementations. Similar to our setting, it treats refinement rules as user-definable artefacts and performs pattern matching to do the refining. Constraints are checked to ensure that the resulting system will be correct. Other work related to B, e.g., [137], is on strictly refining existing functionalities. Approaches described in, e.g., [23, 51, 87, 105] prove that a transformation preserves the semantics of any input model, by showing that the transformed model will be strong or weak bisimilar to the original. Contrary to our work, in all these approaches, no cases can be handled where transformations alter the semantics in a way that does not invalidate the functional property of interest. Furthermore, by using branching bisimilarity our technique also supports abstraction (as opposed to strong) and a subset of liveness properties (as opposed to weak bisimilarity).

Similarly, Combemale *et al.* [51], Hülsbusch *et al.* [105], and Karsai and Narayanan [113, 152] check semantics preservation of model transformations using either strong or weak bisimilarity.

Several techniques perform individual checks for each concrete model [113, 152, 208]. As such, the transformation itself is not verified, but verification is done each time the

[3] `http://www.fsel.com/fdr3.html`.

[4] `http://www.tools.clearsy.com/tools/bart`.

transformation is applied in a concrete situation. Our technique verifies the transformation definition once, after which the verification result is relevant for each application of that transformation.

Monotonically adding functionality, as opposed to refining, is addressed by, e.g., Braunstein and Encrenaz [36]. The focus is on updating property formulae. It would be interesting to investigate whether such an approach can be applied within our technique in order to transform properties along with the system.

In some works, theorem proving is used to verify the preservation of behavioural semantics [86, 194]. The use of theorem provers requires expert knowledge and high effort [194]. In contrast, our equivalence checking approach is more lightweight, automated, and allows the retrieval of counter-examples (as a set of traces) that indicate what behaviour of the right pattern is not branching bisimilar to that of the left pattern. This information helps developers identify issues with the transformations.

Transformation rules for Open Nets are verified on the preservation of dynamic semantics by Baldan et al. [18]. Open Nets are a reactive extension of Petri Nets. The technique is comparable to our technique with two main exceptions. First, they consider weak bisimilarity for the comparison of rule patterns, which preserves a strictly smaller fragment of the μ-calculus than branching bisimilarity [144]. Second, their technique does not allow transforming the communication interfaces between components. Our approach allows this, and checks whether the components remain 'compatible'.

Finally, Selim et al. [187] check correspondence between source and target models for transformations expressed in the DSLTrans. DSLTrans uses a symbolic model checker to verify properties that can be derived from the meta-models. The state space captures the evolution of the input model. In contrast, our approach considers the state spaces of combinations of transformation rules, which represent the potential behaviour described by those rules. The verification of language structural properties offered by DSLTrans and the verification of dynamic semantic properties offered by Refiner address orthogonal aspects of correctness of model transformations. Hence, an interesting pointer for future work is whether those two approaches can be combined.

3.3 Verifying Single LTS Transformations

This section introduces the main concepts related to the transformation of LTSs (Definition 2.2.1), and explains how a single transformation rule can be analysed to guarantee that it preserves the branching structure of all LTSs it can be applied to.

3.3.1 Transformation of LTSs

We allow LTSs to be transformed by means of formally defined transformation rules. Transformation rules are defined as follows.

Definition 3.3.1 (Transformation Rule)**.** *A* transformation rule $r = (\mathcal{L}, \mathcal{R})$ *consists of a left pattern LTS* $\mathcal{L} = (\mathcal{S}_\mathcal{L}, \mathcal{A}_\mathcal{L}, \mathcal{T}_\mathcal{L}, \mathcal{I}_\mathcal{L})$ *and a right pattern LTS* $\mathcal{R} = (\mathcal{S}_\mathcal{R}, \mathcal{A}_\mathcal{R}, \mathcal{T}_\mathcal{R}, \mathcal{I}_\mathcal{R})$*, with* $\mathcal{I}_\mathcal{L} = \mathcal{I}_\mathcal{R}$*. The initial states of the pattern LTSs* $\mathcal{I}_\mathcal{L}$ *and* $\mathcal{I}_\mathcal{R}$ *are called the* in-states*. Furthermore, the two pattern LTSs are annotated with a (possibly empty) set of* exit-states $\mathcal{E}_\mathcal{L} \subseteq \mathcal{S}_\mathcal{L}$ *and* $\mathcal{E}_\mathcal{R} \subseteq \mathcal{S}_\mathcal{R}$*, respectively, with* $\mathcal{E}_\mathcal{L} = \mathcal{E}_\mathcal{R}$*. Finally, we must have that* $\mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R} = \mathcal{I}_\mathcal{L} \cup \mathcal{E}_\mathcal{L} = \mathcal{I}_\mathcal{R} \cup \mathcal{E}_\mathcal{R}$*.*

The states in $\mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R}$ are called the *glue-states*. The *in-states* ($\mathcal{I}_\mathcal{L}$ and $\mathcal{I}_\mathcal{R}$) are glue-states that represent the states at which the pattern may be entered. The *exit-states*

($\mathcal{E}_\mathcal{L}$ and $\mathcal{E}_\mathcal{R}$) are glue-states that represent the states from which the pattern may be left. It is possible for a glue-state to be both an in-state and an exit-state.

Figure 3.2 shows an example of a transformation rule $r = (\mathcal{L}, \mathcal{R})$ transforming a sequence of two a-transitions to a τ-transition followed by two a'-transitions. The initial states, i.e., the in-states of $\mathcal{L}$ and $\mathcal{R}$, are indicated by an incoming arrow. The exit-states are represented by a square. Furthermore, all glue-states (i.e., the in- and exit-states) are coloured grey.

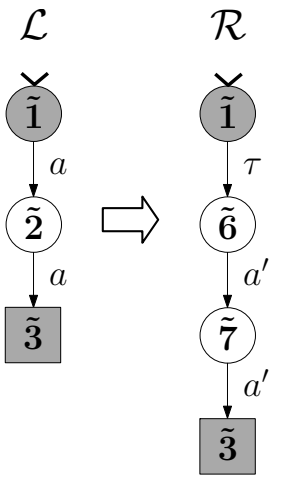


Figure 3.2: A transformation rule

Say the pattern LTS is embedded into a larger LTS. The *context* of the LTS pattern is the behaviour of the larger LTS that is not described in the pattern LTS. The patterns LTS expect only ingoing transitions from the context at state $\langle\tilde{1}\rangle$ as this is an in-state. Similarly, at exit-state $\langle\tilde{3}\rangle$, only outgoing transitions to the context are expected. Our previous formalisation [59] cannot express these subtleties as it does not distinguish between in-states and out-states. In Section 3.3.2, we show that, due to the in-states and out-states, the formalisation presented in this chapter is able to identify that, when the a-transitions are relabelled to a'-transitions, this transformation rule is correct while the previous formalisation cannot.

When applying a transformation rule to an LTS, the changes are applied relative to the glue-states, i.e., the patterns LTS are embeddings in larger LTSs. To reason about the embedding of a transformation rule, we first define the notion of an *LTS morphism*.

Definition 3.3.2 (LTS morphism)**.** *An* LTS morphism $f : \mathcal{G}_0 \to \mathcal{G}_1$ *between two LTSs* $\mathcal{G}_0 = (\mathcal{S}_{\mathcal{G}_0}, \mathcal{A}_{\mathcal{G}_0}, \mathcal{T}_{\mathcal{G}_0}, \mathcal{I}_{\mathcal{G}_0})$ *and* $\mathcal{G}_1 = (\mathcal{S}_{\mathcal{G}_1}, \mathcal{A}_{\mathcal{G}_1}, \mathcal{T}_{\mathcal{G}_1}, \mathcal{I}_{\mathcal{G}_1})$ *is a pair of functions* $f = (f_\mathcal{S} : \mathcal{S}_{\mathcal{G}_0} \to \mathcal{S}_{\mathcal{G}_1}, f_\mathcal{T} : \mathcal{T}_{\mathcal{G}_0} \to \mathcal{T}_{\mathcal{G}_1})$ *which preserve source states, target states, and transition labels, i.e., for all* $s \xrightarrow{a}_{\mathcal{G}_0} s'$*, it holds that* $f_\mathcal{T}(s \xrightarrow{a}_{\mathcal{G}_0} s') = f_\mathcal{S}(s) \xrightarrow{a}_{\mathcal{G}_1} f_\mathcal{S}(s')$.

An embedding of one LTS into another is described by an injective LTS morphism. It should be noted that for such embeddings, there is never a need to explicitly indicate how transitions are mapped by an LTS morphism f as the transition relation uniquely describes tuples consisting of source state, label, and target state. This ensures that given a function $f_\mathcal{S} : \mathcal{S}_{\mathcal{G}_0} \to \mathcal{S}_{\mathcal{G}_1}$, an injective LTS morphism f is implied by it, since no two transitions in $\mathcal{G}_0$ can be mapped to the same transition in $\mathcal{G}_1$, i.e., an injective function $f_\mathcal{T} : \mathcal{T}_{\mathcal{G}_0} \to \mathcal{T}_{\mathcal{G}_1}$ is implied. Because of that, with slight abuse of notation, we directly reason about LTS morphisms f as mappings between LTS states, in the remainder of this chapter.

A transformation rule $r = (\mathcal{L}, \mathcal{R})$ is *applicable* to an LTS $\mathcal{G}$ iff a *match* $m : \mathcal{L} \to \mathcal{G}$ exists according to Definition 3.3.3. Given a state $s \in \mathcal{S}_\mathcal{G}$ of an input LTS $\mathcal{G}$ and a state $p \in \mathcal{S}_\mathcal{P}$ of a pattern LTS $\mathcal{P}$, we write $m(p) = s$ to indicate that state s is *matched on* by state p via match a $m : \mathcal{P} \to \mathcal{G}$. The set $m(S) = \{m(s) \in \mathcal{S}_\mathcal{G} \mid s \in S\}$ is the image of a set of states $S \subseteq \mathcal{S}_\mathcal{P}$ through match m on an LTS $\mathcal{G}$.

Definition 3.3.3 (Match)**.** *A pattern LTS* $\mathcal{P} = (\mathcal{S}_\mathcal{P}, \mathcal{A}_\mathcal{P}, \mathcal{T}_\mathcal{P}, \mathcal{I}_\mathcal{P})$ *with a set of exit-states* $\mathcal{E}_\mathcal{P}$ *has a* match $m : \mathcal{P} \to \mathcal{G}$ *on an LTS* $\mathcal{G} = (\mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G})$ *iff* m *is an injective LTS morphism and for all* $p \in \mathcal{S}_\mathcal{P}, s \in \mathcal{S}_\mathcal{G}$*:*

- $m(p) = s \wedge s \in \mathcal{I}_\mathcal{G} \implies p \in \mathcal{E}_\mathcal{P}$*;*

- $s \xrightarrow{a}_{\mathcal{G}} m(p) \wedge (\neg\exists p' \in \mathcal{S}_{\mathcal{P}}.\ p' \xrightarrow{a}_{\mathcal{P}} p \wedge m(p') = s) \implies p \in \mathcal{I}_{\mathcal{P}}$;
- $m(p) \xrightarrow{a}_{\mathcal{G}} s \wedge (\neg\exists p' \in \mathcal{S}_{\mathcal{P}}.\ p \xrightarrow{a}_{\mathcal{P}} p' \wedge m(p') = s) \implies p \in \mathcal{E}_{\mathcal{P}}$.

A match is a behaviour preserving morphism of a pattern LTS $\mathcal{P}$ in an LTS $\mathcal{G}$ defined via a category of LTSs [222]. The first match condition expresses that an initial state may only be matched on by exit-states. This is a reasonable assumption as all reachable behaviour starts at initial states. A consequence of the condition is that initial states may not be removed by a transformation. If the condition is violated, then both the initial state and its transitions are removed, thus the initial state of the resulting transformed LTS has no transitions either.

The remaining two conditions make sure that a match may not cause removal of transitions that are not explicitly present in $\mathcal{P}$ (an example will be discussed after we introduce the notion of LTS transformation). The first condition ensures that the match of a pattern LTS may only be entered through in-states, i.e., a transition from an unmatched state to a matched state implies that the matched state is matched on by an in-state. Similarly, the second condition states that the match of the pattern LTS may only be left through an exit-state, i.e., a transitions from a matched state to an unmatched state implies that the matched state is matched on by an exit-state.

Figure 3.4 show

An LTS $\mathcal{G}$ is transformed to an LTS $T(\mathcal{G})$ according to Definition 3.3.4. For clarity, we refer with $p, p', \ldots$ to states in a left pattern LTS, with $q, q', \ldots$ to states in a right pattern LTS, with $s, s', \ldots$ to states in an input LTS, and with $t, t', \ldots$ to states in an output LTS.

Definition 3.3.4 (LTS Transformation)**.** *Let $\mathcal{G} = (\mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}})$ be an LTS and let $r = (\mathcal{L}, \mathcal{R})$ be a transformation rule with match $m : \mathcal{L} \to \mathcal{G}$. Moreover, let $\hat{m}$ be the state generating function that will define the match of $\mathcal{R}$ on to the transformed LTS $T(\mathcal{G})$ such that $\forall q \in \mathcal{S}_{\mathcal{L}} \cap \mathcal{S}_{\mathcal{R}}.\ \hat{m}(q) = m(q)$ and $\forall q \in \mathcal{S}_{\mathcal{R}} \setminus \mathcal{S}_{\mathcal{L}}.\ \hat{m}(q) \notin \mathcal{S}_{\mathcal{G}}$. The* transformation *of LTS $\mathcal{G}$, via rule r with matches m, $\hat{m}$, is defined as $T(\mathcal{G}) = (\mathcal{S}_{T(\mathcal{G})}, \mathcal{A}_{T(\mathcal{G})}, \mathcal{T}_{T(\mathcal{G})}, \mathcal{I}_{\mathcal{G}})$ where*

- $\mathcal{S}_{T(\mathcal{G})} = \mathcal{S}_{\mathcal{G}} \setminus m(\mathcal{S}_{\mathcal{L}}) \cup \hat{m}(\mathcal{S}_{\mathcal{R}})$;
- $\mathcal{T}_{T(\mathcal{G})} = (\mathcal{T}_{\mathcal{G}} \setminus \{m(p) \xrightarrow{a} m(p') \mid p \xrightarrow{a}_{\mathcal{L}} p'\}) \cup \{\hat{m}(q) \xrightarrow{a} \hat{m}(q') \mid q \xrightarrow{a}_{\mathcal{R}} q'\}$;
- $\mathcal{A}_{T(\mathcal{G})} = \{a \mid \exists t \xrightarrow{a} t' \in \mathcal{T}_{T(\mathcal{G})}\}$.

Given a match, an LTS transformation replaces states and transitions matched by $\mathcal{L}$ by a copy of $\mathcal{R}$ yielding LTS $T(\mathcal{G})$. An application of a transformation rule is shown in Figure 3.3. Again, the initial states are indicated by an incoming arrow. In the middle of Figure 3.3, the transformation rule $r = (\mathcal{L}, \mathcal{R})$ is shown (presented earlier in Figure 3.2) which is applied to LTS $\mathcal{G}$ resulting in LTS $T(\mathcal{G})$. The states are numbered such that matches can be identified by the state label, i.e., a state $\tilde{i}$ is matched onto state i. Note that such a match satisfies the conditions of Definition 3.3.3: State $\langle\tilde{1}\rangle$ is not an exit-state, but state $\langle 1\rangle$ does not have unmatched outgoing transitions, state $\langle\tilde{3}\rangle$ is not an in-state, but there are no unmatched incoming transitions to state $\langle 3\rangle$, and finally, state $\langle 3\rangle$ has unmatched outgoing transitions, but this is allowed, since $\langle\tilde{3}\rangle$ is an exit-state.

On the other hand, states $\langle\tilde{1}\rangle$, $\langle\tilde{2}\rangle$ and $\langle\tilde{3}\rangle$ of $\mathcal{L}$ do not match on states $\langle 2\rangle$, $\langle 3\rangle$, and $\langle 4\rangle$, respectively, as this violates condition 2 of Definition 3.3.3. Namely, transition

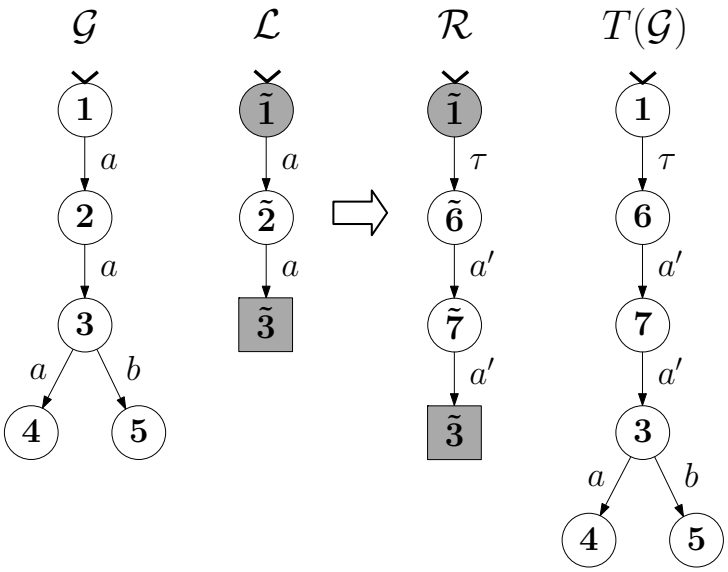


Figure 3.3: Application of a transformation rule

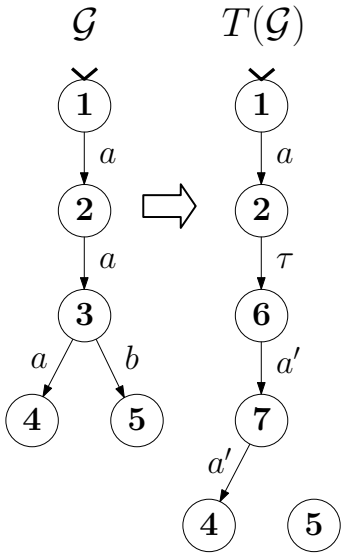


Figure 3.4: Violation of the conditions of a match results in the deletion of the b-transition

$\langle 3 \rangle \xrightarrow{b}_{\mathcal{G}} \langle 5 \rangle$ is unmatched, since state $\langle 5 \rangle$ is unmatched, but state $\langle 2 \rangle$ is not an exit-state. Figure 3.4 shows what happens if the transformation is applied using this morphism despite it satisfying the match conditions. When $\langle 2 \rangle$ is deleted by the transformation its b-transition to state $\langle 4 \rangle$ no longer has a source state, and hence, is deleted as well. The conditions of a match prevent the occurrence of such undesirable situations.

In general, $\mathcal{L}$ may have several matches on $\mathcal{G}$. Therefore, we assume that transformations are *confluent*, i.e., that they are guaranteed to terminate and lead to a unique $T(\mathcal{G})$. Confluence of LTS transformations can be checked efficiently [215]. By assuming confluence, we can focus on having a single match when verifying transformation rules, since the transformations of individual matches do not influence each other.

The transformation shown in Figure 3.3 is confluent. The transformation rule can be made non-confluent by changing state $\langle \tilde{2} \rangle$ to an exit-state. Then, the transformation rule is also applicable to $\langle 2 \rangle \xrightarrow{a} \langle 3 \rangle \xrightarrow{a} \langle 4 \rangle$. Hence, such a transformation can lead to two different transformed LTSs that cannot be transformed further. Not only is easier to understand confluent transformations leading to a single unique transformed LTS, we also exploit this property in Section 3.4.3 to weaken some of the preconditions that the technique relies upon for verification of sets of dependent transformation rules.

3.3.2 Analysing a Transformation Rule

The basis of the transformation verification procedure is to check whether the two patterns making up a transformation rule are equivalent, while respecting that these patterns represent embeddings in larger systems. We want to be able to verify the transformation's side effects on both the matched states and the states connected to these matched states. To make this explicit, we extend the left- and right-patterns of a transformation rule $r = (\mathcal{L}, \mathcal{R})$ according to Definition 3.3.5. The resulting so-called κ-*extended* transformation rule is defined as $r^\kappa = (\mathcal{L}^\kappa, \mathcal{R}^\kappa)$, and is specifically used for the purpose of analysing r, it does not replace r.

In the κ-extended version of a pattern LTS $\mathcal{P}$, a new state named κ is introduced,

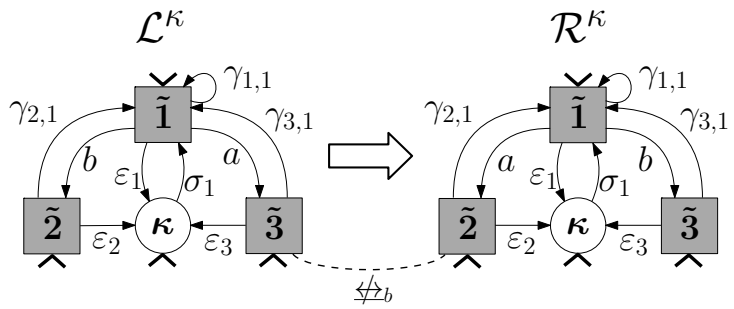


Figure 3.5: The ε_3-transition ensures $\langle 3 \rangle \not\leftrightarrow_b \langle 2 \rangle$

which is connected to the original states by new transitions labelled σ_p for $p' \in \mathcal{I}_\mathcal{P}$, and $\varepsilon_{p'}$ for $p \in \mathcal{E}_\mathcal{P}$. Furthermore, for all $p \in \mathcal{E}_\mathcal{P}$ and $p' \in \mathcal{I}_\mathcal{P}$ a $\gamma_{p,p'}$-transition is introduced. The set of initial states of the κ-extended LTS consists of the states in $\mathcal{E}_\mathcal{P}$ extended with a single and unique κ-state. The in-states $p \in \mathcal{I}_\mathcal{P}$ do not need to be added to the set of initial states as they are always reachable via a σ-transition.

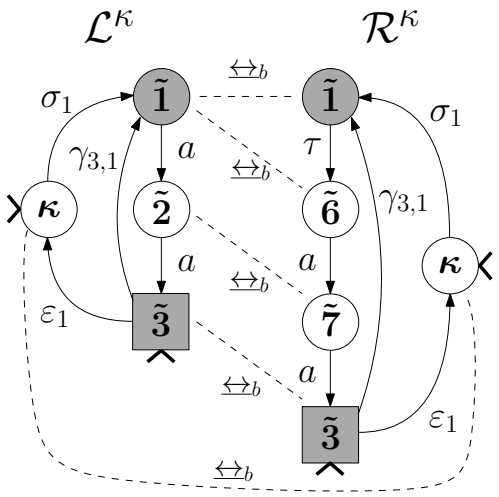
Definition 3.3.5 (κ-extension of a pattern LTS)**.** *The pattern LTS $\mathcal{P}$ extended with a κ-state, and σ-, ε- and γ-transitions is defined as $\mathcal{P}^\kappa = (\mathcal{S}_{\mathcal{P}^\kappa}, \mathcal{A}_{\mathcal{P}^\kappa}, \mathcal{T}_{\mathcal{P}^\kappa}, \mathcal{I}_{\mathcal{P}^\kappa})$ where*

- $\mathcal{S}_{\mathcal{P}^\kappa} = \mathcal{S}_\mathcal{P} \cup \{\kappa\}$;
- $\mathcal{A}_{\mathcal{P}^\kappa} = \mathcal{A}_\mathcal{P} \cup \{\sigma_p \mid p \in \mathcal{I}_\mathcal{P}\} \cup \{\varepsilon_p \mid p \in \mathcal{E}_\mathcal{P}\} \cup \{\gamma_{p,p'} \mid p \in \mathcal{E}_\mathcal{P} \wedge p' \in \mathcal{I}_\mathcal{P}\}$;
- $\mathcal{T}_{\mathcal{P}^\kappa} = \mathcal{T}_\mathcal{P} \cup \{\kappa \xrightarrow{\sigma_p} p \mid p \in \mathcal{I}_\mathcal{P}\} \cup \{p \xrightarrow{\varepsilon_p} \kappa \mid p \in \mathcal{E}_\mathcal{P}\} \cup \{p \xrightarrow{\gamma_{p,p'}} p' \mid p \in \mathcal{E}_\mathcal{P} \wedge p' \in \mathcal{I}_\mathcal{P}\}$;
- $\mathcal{I}_{\mathcal{P}^\kappa} = \{\kappa\} \cup \mathcal{E}_\mathcal{P}$

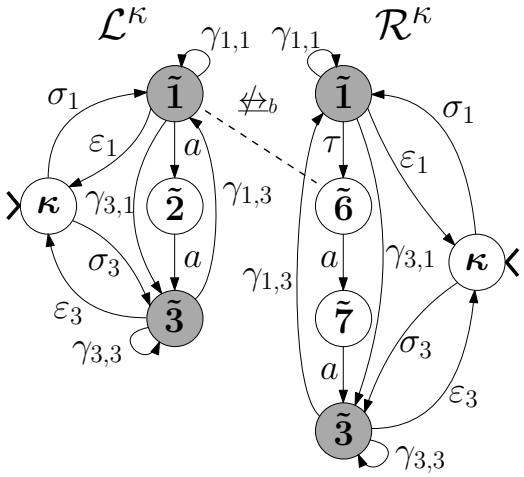
with $\mathcal{E}_{\mathcal{P}^\kappa} = \mathcal{E}_\mathcal{P}$ and where σ_p, ε_p and $\gamma_{p,p'}$ are unique labels that are not the silent τ-label.

The κ-extension of an LTS pattern $\mathcal{P}$ can be seen as an abstraction of LTSs it is matched on, in which we indicate how the behaviour described by $\mathcal{P}$ can be embedded in a larger LTS $\mathcal{G}$. The introduced κ-state represents the unmatched (and thus unaffected) states in $\mathcal{G}$. The σ-transitions go from the κ-state to the in-states and represent transitions that enter the part of $\mathcal{G}$ matched on by $\mathcal{P}$. The ε-transitions go from exit-states to the κ-state. They represent transitions in $\mathcal{G}$ that leave the part matched on by $\mathcal{P}$. The γ-transitions go from exit-states to in-states representing transitions connected to states that are matched on $\mathcal{P}$, while the transition itself is not matched on. The σ-, ε-, and γ-transitions are uniquely identified by their corresponding glue-states. This ensures that side effects on unmatched states become visible.

If the original $\mathcal{L}$ and $\mathcal{R}$ are branching bisimilar (Definition 2.2.2), then one cannot in general conclude that input and output LTSs to which the rule $r = (\mathcal{L}, \mathcal{R})$ is applicable are branching bisimilar as well. For instance, consider the transformation rule in Figure 3.5 which swaps a and b transitions. Without the κ-extensions, the LTS patterns are branching bisimilar. However, this would not capture the fact that patterns should be interpreted as possible embeddings in larger LTSs. These larger LTSs may not be branching bisimilar, because glue-states $\langle \tilde{2} \rangle$ and $\langle \tilde{3} \rangle$ could be mapped to states with different outgoing transitions, apart from the behaviour described in the LTS patterns (states $\langle \tilde{2} \rangle$ and $\langle \tilde{3} \rangle$ are exit-states). However, due to the introduced κ-state and in



(a) The κ-extension of the transformation rule shown in Figure 3.2, relabelled with $a' := a$, using the formalisation in this chapter; the left and right κ-extended pattern LTSs are branching bisimilar.



(b) The κ-extension of the transformation rule shown in Figure 3.2, relabelled with $a' := a$, where in-states and exit-states are not distinguished from each other; the left and right κ-extended pattern LTSs are *not* branching bisimilar since in $\mathcal{R}^\kappa$, the possibility of performing a ε_1-transition is lost once the τ-transition from state $\langle\tilde{1}\rangle$ to state $\langle\tilde{6}\rangle$ is taken.

Figure 3.6: The approach presented in this chapter (Figure 3.6a) is able to determine that the transformation rule shown in Figure 3.2 (where a' has been relabelled to a) guarantees that the input and output LTSs are branching bisimilar; this is an improvement over our previous formalisation [59] (Figure 3.6b) which reports a counter-example as it does not distinguish between in- and exit-states.

particular the ε-transitions, a comparison of the κ-extended networks is able to determine that the rule does not guarantee branching bisimilarity between input and output LTSs.

Figure 3.6 shows that the verification approach discussed in this chapter is able to perform a more fine grained analysis compared to the approach in previous work [59]. The κ-extension of the transformation rule in Figure 3.2, but now with a' replaced by a, is shown in Figure 3.6a and Figure 3.6b using the approach presented in this chapter and the approach in previous work, respectively. In the latter case, the notions of in-state and exit-state are not used, instead both types of states are treated in the same way, as glue-states.

The approach discussed in this chapter determines that the left and right κ-extended pattern LTSs are branching bisimilar as shown in Figure 3.6a. The branching bisimulation relation between the left and right κ-extended pattern LTSs is indicated with dashed lines. The introduction of the τ-transition does not break branching bisimilarity since no behaviour is lost.

However, the approach in [59] reports a counter-example as shown in Figure 3.6b. Since the approach in [59] does not distinguish between in-states and exit-states, the semantics of the transformation rule is slightly different; each glue-state is allowed to be matched on states with ingoing transitions, outgoing transitions, and both in- and outgoing transitions. Therefore, any correct verification technique would have to consider the possibility that the glue-states are matched on states that have additional in- and/or outgoing transitions, and

therefore, the extra τ-transition in $\mathcal{R}^\kappa$ could mean that unmatched outgoing transitions are disabled when the τ-transition is followed. By adding the notions of in-state and exit-state, we can restrict the applicability of transformation rules and thereby provide more information to the verification technique.

The analysis In the *verification* of a *transformation rule* $r = (\mathcal{L}, \mathcal{R})$ the aim is to determine whether r is sound for any LTS $\mathcal{G}$ to which r is applicable. The verification proceeds as follows:

1. Construct the κ-extended pattern LTSs $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ according to Definition 3.3.5.
2. Determine whether $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ are branching bisimilar.

If $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ are branching bisimilar, then r is branching-structure preserving for all inputs it is applicable on. Otherwise, r may preserve the branching-structure of some LTSs, but it is definitely not branching-structure preserving for *all possible* inputs it is applicable to.

Time complexity of the analysis Consider a transformation rule r. Let g be the number of glue-states defined in the pattern LTSs of r. Furthermore, let s, t and a be the largest number of states, transitions and action labels, respectively in the pattern LTSs of r.

In the *first* step of the verification of a rule r, a κ-state is added, i.e., the number of states added is constant. In the worst case, all glue-states are both in and exit-states. For each glue state one σ- and ε-transition is added, i.e., $O(g)$ σ- and ε-transitions are added. Furthermore, there are as many γ-transitions as combinations of σ- and ε-transitions, i.e., $O(g^2)$ γ-transitions are added. Finally, the number of action labels added is $O(g^2)$ because of the quadratic amount of added γ-labels. Hence, the running time of step 1 is $O(g^2)$.

In the *second* step, it is checked whether $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ are branching bisimilar. Branching bisimilarity checking can be performed in $O(t \cdot \log(s + a))$ [91]. Therefore, the time complexity of the final step of the analysis is $O((t + g^2) \cdot \log((s + 1) + (a + g^2)))$.

3.3.3 Correctness of the verification

In this section we prove the correctness of the analysis algorithm presented in the previous section. First, we introduce two lemmas that express properties of left and right κ-extended pattern LTSs that are branching bisimilar. Next, we prove the soundness of the approach in Proposition 3.3.9. Finally, the completeness of the approach is proven in Proposition 3.3.10.

Recall that glue-states are not removed by transformation and that the κ-state represents unmatched states, and as κ-states remain unchanged, they also represent states that are not removed. When comparing LTS patterns by checking for branching bisimilarity, it is desirable that these states are related to themselves, as illustrated in the previous example. Lemma 3.3.6 shows that it is indeed the case that κ-extension achieves this: if two κ-extended pattern LTSs $\mathcal{L}^\kappa$, $\mathcal{R}^\kappa$ are branching bisimilar, then the κ-state, the in-states, and the exit-states, i.e., the initial states of the κ-extended LTS patterns, are related to themselves.

Lemma 3.3.6. *Consider a transformation rule $r = (\mathcal{L}, \mathcal{R})$ and a branching bisimulation relation B between $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$. Then, $\forall p \in \{\kappa\} \cup \mathcal{I}_\mathcal{L} \cup \mathcal{E}_\mathcal{L}, p\ B\ p$.*

Proof. The proof follows from the fact that the σ- and ε-transitions are uniquely constructed for a specific glue-state. Consider a state $p \in \{\kappa\} \cup \mathcal{I}_\mathcal{L} \cup \mathcal{E}_\mathcal{L}$. By Definition 3.3.5, we have $p \in \mathcal{I}_{\mathcal{L}^\kappa}$. Since $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ are branching bisimilar, there is a state $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $p\ B\ q$. We perform a case distinction on $p \in \{\kappa\} \cup \mathcal{I}_\mathcal{L} \cup \mathcal{E}_\mathcal{L}$. In each case we show that there is a transition labelled with a σ or ε between p and $p' \in \mathcal{S}_{\mathcal{L}^\kappa}$ such that the action label uniquely identifies the states p and p'. For convenience, let us refer to this unique label as α and say we have a transition $p \xrightarrow{\alpha}{}^{\kappa}_{\mathcal{L}}\ p'$. As $p\ B\ q$ we can apply Definition 2.2.2 to show that q simulates p. As the unique labels are not allowed to be the silent action τ, the only remaining case indicates that there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}{}^*_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\alpha}_{\mathcal{R}^\kappa} q'$ with $p\ B\ \hat{q}$ and $p'\ B\ q'$. There is only one transition labelled α in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ and it occurs as $p \xrightarrow{\alpha} p'$. It follows that $\hat{q} = p$ and $q' = p'$. Consequently, we have $p\ B\ p$ and $p'\ B\ p'$.

We now discuss the case distinction in full detail:

- $p = \kappa$. By Definition 2.2.1, $\mathcal{I}_\mathcal{L} \neq \emptyset$, so there is a state $p' \in \mathcal{I}_\mathcal{L}$. This means that there is a transition $\kappa \xrightarrow{\sigma_{p'}}_{\mathcal{L}^\kappa} p'$ where $\sigma_{p'} \neq \tau$ and $\sigma_{p'}$ uniquely occurs on $\kappa \xrightarrow{\sigma_{p'}} p'$ in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ (Definition 3.3.5). Hence, since $\sigma_{p'} \neq \tau$, by Definition 2.2.2, there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}{}^*_{\mathcal{R}} \hat{q} \xrightarrow{\sigma_{q'}}_{\mathcal{R}} q'$ with $\kappa\ B\ \hat{q}$.The $\sigma_{p'}$-transition in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ is strictly present as $\kappa \xrightarrow{\sigma_{p'}} p'$. It follows that $\hat{p} = \kappa$, therefore we have $p\ B\ p$.

- $p \in \mathcal{I}_\mathcal{L}$. Then there is a transition $\kappa \xrightarrow{\sigma_p}_{\mathcal{L}^\kappa} p$ where $\sigma_p \neq \tau$ and σ_p uniquely occurs from κ to p in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ (Definition 3.3.5). In the previous case we established that $\kappa\ B\ \kappa$. Because $\sigma_p \neq \tau$, by Definition 2.2.2, we have states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $\kappa \xrightarrow{\tau}{}^*_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\sigma_p}_{\mathcal{R}^\kappa} q'$ with $p\ B\ q'$. The σ_p-transition in $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$ only goes from κ to p. It follows that $q' = p$ and thus $p\ B\ p$.

- $p \in \mathcal{E}_\mathcal{L}$. By Definition 3.3.5, there is a state $p \in \mathcal{E}_{\mathcal{L}^\kappa}$ with an observable action ε_p that uniquely occurs on a transition from p to κ in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$. Moreover, since B relates $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$, there is a state $q \in \mathcal{I}_{\mathcal{R}^\kappa}$ such that $p\ B\ q$. Therefore, by $\varepsilon_p \neq \tau$ and Definition 2.2.2, there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}{}^*_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\varepsilon_p}_{\mathcal{R}} q'$ with $p\ B\ \hat{q}$ and $\kappa\ B\ q'$. By Definition 3.3.5, there is only one transition labelled ε_p in $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$, which goes from p to κ. It follows that $\hat{q} = p$ and hence $p\ B\ p$. □

Exit-states are the states where the embedding of an LTS pattern may be left. A transition leaving the embedding is represented in the κ-extended pattern by an ε-transition. Should an arbitrary state $q \in \mathcal{S}_\mathcal{R}$ be related to an exit-state $p \in \mathcal{E}_\mathcal{L}$, then there must exist a τ-path from q to p, otherwise state q cannot simulate the ε-transitions from p. Lemma 3.3.7 shows that, indeed, such a τ-path from q to p exists.

Lemma 3.3.7. *Consider a transformation rule $(\mathcal{L}, \mathcal{R})$ and a branching bisimulation relation B between $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$, then*

$$\forall p \in \mathcal{E}_{\mathcal{L}^\kappa}, q \in \mathcal{S}_{\mathcal{R}^\kappa} : p\ B\ q \implies (q \xrightarrow{\tau}{}^*_{\mathcal{R}} p)$$

Proof. Intuitively, the proof follows from the fact that action ε_p uniquely occurs on a transition from p to κ. Since in $\mathcal{L}^\kappa$, any state q branching bisimilar to p must be able to perform this transition directly or be able to reach such a transition via a τ-path, we must either have that $q = p$ or that from q, p can be reached via a τ-path. Next, we discuss the proof in full detail.

Let state $p \in \mathcal{E}_\mathcal{L}$ and state $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \; B \; p$. By Definition 3.3.5 we have $p \xrightarrow{\varepsilon_p}_\mathcal{L} \kappa$ with $\varepsilon_p \neq \tau$. Since $p \; B \; q$ and $\varepsilon_p \neq \tau$ there are $\hat{q}, q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}{}^*_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\varepsilon_p}_{\mathcal{R}^\kappa} q'$ with $p \; B \; \hat{q}$ and $\kappa \; B \; q'$. The ε_p action only occurs on $p \xrightarrow{\varepsilon_p} \kappa$ in both $\mathcal{L}^\kappa$ and $\mathcal{R}^\kappa$, therefore, we must have $\hat{q} = p$ and $q' = \kappa$. It follows that $q \xrightarrow{\tau}{}^*_{\mathcal{R}^\kappa} p$ and by structural induction $q \xrightarrow{\tau}{}^*_\mathcal{R} p$. □

Definition 3.3.8 introduces a mapping that formally defines how a κ-extended LTS pattern is related to the LTS that is matched on. The fact that κ-states represent all states that are not matched on is made explicit by this mapping.

Definition 3.3.8 (Mapping of κ-extended LTS)**.** *Consider an LTS $\mathcal{G}$ and a pattern LTS $\mathcal{P}$ with corresponding match $m : \mathcal{P} \to \mathcal{G}$. We say that a $p \in \mathcal{S}_{\mathcal{P}^\kappa}$ is* mapped *to a state $s \in \mathcal{S}_\mathcal{G}$, denoted by $m^\kappa(p) = s$, iff either $p \neq \kappa$ and $m(p) = s$ or $p = \kappa$ and there is no state in $\mathcal{S}_\mathcal{L}$ matching on s (i.e., $\neg\exists x \in \mathcal{S}_\mathcal{P}, m(x) = s$).*

Soundness of the analysis A transformation rule r preserves the branching structure of all LTSs it is applicable to if the κ-extended patterns of r are branching bisimilar. This is expressed in Proposition 3.3.9.

Proposition 3.3.9 is a special case of Proposition 3.4.16, discussed in the next section where transformations of concurrent systems are considered. This proof is derived from the Coq proof of Proposition 3.4.16 to explain the transformation verification technique in a simpler and more intuitive setting.

Proposition 3.3.9. *Let $\mathcal{G}$ be an LTS, let r be a transformation rule with matches $m : \mathcal{L} \to \mathcal{G}$ and $\hat{m} : \mathcal{R} \to T(\mathcal{G})$ such that Definition 3.3.4 is satisfied. Then,*

$$\mathcal{L}^\kappa \leftrightarrow\!\!\!\!\!\!\underline{}_b \; \mathcal{R}^\kappa \implies \mathcal{G} \leftrightarrow\!\!\!\!\!\!\underline{}_b \; T(\mathcal{G})$$

Intuition. A match of pattern $\mathcal{L}$ is replaced with an instance of pattern $\mathcal{R}$. If $\mathcal{L}^\kappa \leftrightarrow\!\!\!\!\!\!\underline{}_b \; \mathcal{R}^\kappa$, then these two patterns exhibit branching bisimilar behaviour, even when they are embedded into a larger LTS. Therefore, the behaviour of the original and transformed system ($\mathcal{G}$ and $T(\mathcal{G})$, respectively) are branching bisimilar.

Proof. By definition, we have $\mathcal{G} \leftrightarrow\!\!\!\!\!\!\underline{}_b \; T(\mathcal{G})$ iff $\mathcal{I}_\mathcal{G} \leftrightarrow\!\!\!\!\!\!\underline{}_b \; \mathcal{I}_{T(\mathcal{G})}$, which means that there must exist a branching bisimulation relation C relating the states in $\mathcal{I}_\mathcal{G}$ and $\mathcal{I}_{T(\mathcal{G})}$. Let B be a branching bisimulation relation demonstrating that $\mathcal{L}^\kappa \leftrightarrow\!\!\!\!\!\!\underline{}_b \; \mathcal{R}^\kappa$. Relation C is constructed as follows:

$$\begin{aligned} C = \{(s,t) \mid & \exists p \in \mathcal{S}_{\mathcal{L}^\kappa}, q \in \mathcal{S}_{\mathcal{R}^\kappa}.\; p \; B \; q \wedge m^\kappa(p) = s \wedge \hat{m}^\kappa(q) = t \\ & \wedge ((p = \kappa \vee q = \kappa) \implies s = t)\} \end{aligned}$$

States that are touched by the transformation are related via the corresponding matches m and $\hat{m}$, and branching bisimulation relation B. States that are left untouched by the transformation are represented by the κ-state for which it holds that $\kappa \; B \; \kappa$

(Lemma 3.4.9). The mappings m^κ and $\hat{m}^\kappa$ map the κ-state on all states that are not matched on. To ensure that untouched states are related to themselves we require $s = t$ whenever either s or t is mapped on by a κ-state.

We now prove that C is a branching bisimulation relation by showing that the initial states of $\mathcal{G}$ and $T(\mathcal{G})$ are related, and that Definition 2.2.2 holds for C. For the latter we only discuss one of the two symmetric cases.

- *C relates the initial states of $\mathcal{G}$ and $T(\mathcal{G})$.* Since we have $\mathcal{I}_\mathcal{G} = \mathcal{I}_{T(\mathcal{G})}$ we only have to show $\forall s \in \mathcal{I}_\mathcal{G}.\ \exists t \in \mathcal{I}_\mathcal{G}.\ s\ C\ t$. Take s again for t, we have to show that $s\ C\ s$. State s is either matched on or not matched on:
  - s is matched on by m, i.e., $\exists p \in \mathcal{S}_\mathcal{L}.\ m(p) = s$. We have $p \in \mathcal{E}_\mathcal{L}$ since initial states may only be matched on by exit-states (first condition of Definition 3.3.3). By Lemma 3.3.6 it follows that $p\ B\ p$. Hence, we have $s\ C\ s$.
  - s is not matched on by m, $\neg\exists p \in \mathcal{S}_\mathcal{L}.\ m(p) = s$. By definition we have $m^\kappa(\kappa) = s$. By Lemma 3.3.6 it follows that $\kappa\ B\ \kappa$. Therefore, we have $s\ C\ s$.

  In both cases it holds that $s\ C\ s$.

- *If $s\ C\ t$ and $s \xrightarrow{a}_\mathcal{G} s'$ then either $a = \tau \wedge s'\ C\ t$, or $t \xrightarrow{\tau}{}^*_{T(\mathcal{G})} \hat{t} \xrightarrow{a}_{T(\mathcal{G})} t' \wedge s\ C\ \hat{t} \wedge s'\ C\ t'$.* By definition of $s\ C\ t$, there are states $p \in \mathcal{S}_{\mathcal{L}^\kappa}$ and $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $p\ B\ q$, $m^\kappa(p) = s$, $\hat{m}^\kappa(q) = t$, and $(p = \kappa \vee q = \kappa) \Rightarrow s = t$ (3.1) . Furthermore, by definition of m^κ, there is a state $p' \in \mathcal{S}_{\mathcal{L}^\kappa}$ such that $m^\kappa(p') = s'$. The transition $s \xrightarrow{a}_\mathcal{G} s'$ is either matched on by a transition $p \xrightarrow{a}_\mathcal{L} p'$ or not match on:
  1. *There exists a transition $p \xrightarrow{a}_\mathcal{L} p'$ matching on $s \xrightarrow{a}_\mathcal{G} s'$ in $\mathcal{L}$.* Since $p\ B\ q$, by Definition 2.2.2, we have the following two cases:
     - *$a = \tau$ with $p'\ B\ q$.* Since $m^\kappa(p') = s'$ and $\hat{m}^\kappa(q) = t$, we have $s'\ C\ t$.
     - *$q \xrightarrow{\tau}{}^*_\mathcal{R} \hat{q} \xrightarrow{a}_\mathcal{R} q'$ with $p\ B\ \hat{q}$ and $p'\ B\ q'$.* Transitions from and to κ-states are not matched on by m and $\hat{m}$, i.e., only transitions in $\mathcal{T}_\mathcal{L}$ ($\mathcal{T}_\mathcal{R}$) match on transitions in $\mathcal{T}_\mathcal{G}$ ($\mathcal{T}_{T(\mathcal{G})}$). Hence, states $p, p', q, \hat{q}$ and q' cannot be κ-states. It follows that $s\ C\ \hat{m}^\kappa(\hat{q})$ and $s'\ C\ \hat{m}^\kappa(q')$, since the matching states are not κ, $m^\kappa(p) = s$, and $m^\kappa(p') = s'$. Finally, as $\hat{m}^\kappa(q) = t$, we have $t \xrightarrow{\tau}{}^*_{T(\mathcal{G})} \hat{m}^\kappa(\hat{q}) \xrightarrow{a}_{T(\mathcal{G})} \hat{m}^\kappa(q')$.
  2. *There is no transition matching $s \xrightarrow{a}_\mathcal{G} s'$ in $\mathcal{L}$, i.e., $\neg p \xrightarrow{a}_\mathcal{L} p'$.* Thus, both s and s' are not removed by the transformation. We distinguish two cases:
     - *State s is not matched on by m.* Therefore, we must have $m^\kappa(p) = s$ with $p = \kappa$. It now follows from (3.1) that $s = t$. Hence, $t \xrightarrow{a}_{T(\mathcal{G})} s'$, and by reflexivity of $\xrightarrow{\tau}{}^*$, $t \xrightarrow{\tau}{}^*_{T(\mathcal{G})} t \xrightarrow{a}_{T(\mathcal{G})} s'$.
       We have $s\ C\ t$, thus, what remains to be shown is $s'\ C\ s'$. Since s is not matched on, it follows from Definition 3.3.3 that state s' is either not matched on or matched on by an in-state. In the former case we have $p' = \kappa$, and in the latter case we have $p' \in \mathcal{I}_\mathcal{L}$. In both cases we can apply Lemma 3.3.6 to obtain $p'\ B\ p'$. It follows that $s'\ C\ s'$.
     - *State s is matched on by a state p, i.e., $m(p) = s$.* We must have $p \neq \kappa$. Since there is no transition matching $s \xrightarrow{a}_{\mathcal{T}_\mathcal{G}} s'$, it follows from the second matching condition (Definition 3.3.3) that $p \in \mathcal{E}_\mathcal{L}$. Now it follows from $p\ B\ q$ and Lemma 3.3.7 that $q \xrightarrow{\tau}{}^*_\mathcal{R} p$. Moreover, since $\hat{m}$ is an embedding the transition is preserved in $T(\mathcal{G})$ and we have $t \xrightarrow{\tau}{}^*_{T(\mathcal{G})} s \xrightarrow{a}_{T(\mathcal{G})} s'$.

What is left to show is that $s\ C\ s$ and $s'\ C\ s'$. As $p \in \mathcal{E}_{\mathcal{L}}$ and there is no transition in $\mathcal{L}$ matching $s \xrightarrow{a}_{\mathcal{G}} s'$ the state q' must be either a κ-state or an in-state, i.e., $p' \in \mathcal{I}_{\mathcal{L}} \cup \{\kappa\}$. For $p \in \mathcal{E}_{\mathcal{L}}$ and $p' \in \mathcal{I}_{\mathcal{L}} \cup \{\kappa\}$ it follows from Lemma 3.3.6 that $p\ B\ p$ and $p'\ B\ p'$, respectively. Hence, we have $s\ C\ t$ and $s'\ C\ s'$.

- *If $s\ C\ t$ and $t \xrightarrow{a}_{T(\mathcal{G})} t'$ then either $a = \tau \wedge s\ C\ t'$, or $s \xrightarrow{\tau}{}^*_{\mathcal{G}} \hat{s} \xrightarrow{a}_{\mathcal{G}} s' \wedge \hat{s}\ C\ t \wedge s'\ C\ t'$.* This case is symmetric to the previous case. □

Completeness of the analysis Completeness is an important factor in verification. A complete analysis technique will not report false negatives. The next proposition expresses that our analysis technique is complete. In the context of this work, completeness means that the analysis will always report that the left and right κ-extended pattern LTSs of a transformation rule r are branching bisimilar if the input LTS $\mathcal{G}$ and output LTS $T(\mathcal{G})$ produced by applying r to $\mathcal{G}$ are branching bisimilar for any given input LTS $\mathcal{G}$ and any given matching. The proof for Proposition 3.3.10 is derived from the Coq proof of Proposition 3.4.17 to explain the transformation verification technique in a simpler and more intuitive setting.

We want to stress that the analysis considers all possible input LTSs. Because of this, it may be that the analysis reports that a transformation rule does not preserve a given property in general, while the property may still hold after transformation of some specific input LTS. Consider, for instance, a transformation rule r that is *not* property preserving according to the analysis. There may still be an input LTS $\mathcal{G}_1$ with a match m_1 such that $\mathcal{G}_1 \underline{\leftrightarrow}_b T(\mathcal{G}_1)$. However, it is guaranteed that there also exists an LTS $\mathcal{G}_2$ for r with a corresponding match m_2 for which $\mathcal{G}_2 \not\underline{\leftrightarrow}_b T(\mathcal{G}_2)$.

Proposition 3.3.10. *Consider a transformation rule $r = (\mathcal{L}, \mathcal{R})$. Let $\mathbb{G}$ be the set of all LTSs. Let $r_{\mathcal{G}}$ be the set of all possible match pairs corresponding to a transformation of an LTS $\mathcal{G}$ where $r_{\mathcal{G}}$ consists of tuples of the form $(m : \mathcal{L} \to \mathcal{G}, \hat{m} : \mathcal{R} \to T(\mathcal{G}))$. The following holds*

$$(\forall \mathcal{G} \in \mathbb{G}, (m, \hat{m}) \in r_{\mathcal{G}}.\ \mathcal{G} \underline{\leftrightarrow}_b T(\mathcal{G})) \implies \mathcal{L}^{\kappa} \underline{\leftrightarrow}_b \mathcal{R}^{\kappa}$$

Proof. Assume that for all $\mathcal{G} \in \mathbb{G}$ and $(m, \hat{m}) \in r_{\mathcal{G}}$ it holds that $\mathcal{G} \underline{\leftrightarrow}_b T(\mathcal{G})$. Trivially, we have $\mathcal{L}^{\kappa} \in \mathbb{G}$ and trivial matches $m : \mathcal{L} \to \mathcal{L}^{\kappa}$, $\hat{m} : \mathcal{R} \to T(\mathcal{L}^{\kappa})$. It follows from the assumption that $\mathcal{L}^{\kappa} \underline{\leftrightarrow}_b T(\mathcal{L}^{\kappa})$. Moreover, by Definition 3.3.4, $T(\mathcal{L}^{\kappa}) = \mathcal{R}^{\kappa}$. It follows that $\mathcal{L}^{\kappa} \underline{\leftrightarrow}_b \mathcal{R}^{\kappa}$. □

3.4 Verifying Sets of Dependent LTS Transformations

In this section, we extend the setting by considering *sets* of interacting process LTSs in so-called *networks of LTSs* [133], or *LTS networks* for short (Definition 2.3.1). Transformations can now affect multiple LTSs in an input network, and the analysis of transformations is more involved, since changes to process-local behaviour may affect system-global properties. Finally, we prove the correctness of the technique based on the complete Coq proof. From the proof it follows that per set of related transformation rules, only a single bisimulation check is required in order to verify a system of transformation rules.

3.4.1 Transformation of LTS Networks

A *system of transformation rules*, or a *rule system* for short, allows the transformation of LTS networks. A rule system transforms multiple processes and may introduce new synchronisation laws. In this chapter, we only consider transformation of admissible LTS networks (Definition 2.3.2). The rule system is defined as follows.

Definition 3.4.1 (Rule system)**.** *A* rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ *consists of a vector of transformation rules* R*, a set of synchronisation laws* $\mathcal{V}'$ *that must be present in the networks that* Σ *is applied to, and a set of synchronisation laws* $\hat{\mathcal{V}}$ *introduced in the network resulting from a transformation. The* i^{th} *left and right pattern LTSs of* R *are denoted by* $\mathcal{L}_i$ *and* $\mathcal{R}_i$*, respectively.*

Intuitively, a rule system describes how a concurrent system is modified to create a transformed concurrent system. A rule system is designed with a specific result in mind. Therefore, it is desirable that a rule system is *confluent* such that transformation rules can be applied in any order eventually leading to the same result. Checking confluence can be done efficiently [215]. In the remainder of this chapter, we only consider confluent rule systems.

The *transformation* of an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of size n given a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ is achieved via a set $\mathfrak{m}$ of pairs of matches. Each element $(m, \hat{m}) \in \mathfrak{m}$ corresponds to the application of some rule in R to a process in Π. The match $m : \mathcal{L}_j \to \Pi_i$ matches the left pattern LTS ($\mathcal{L}_j$) of the j^{th} transformation rule in R on to the i^{th} process LTS (Π_i) of LTS network $\mathcal{N}$. Similarly, match $\hat{m} : \mathcal{R}_j \to T(\Pi_i)$ matches the right process LTS ($\mathcal{R}_j$) of rule R_j in R on to the transformed process LTS ($T(\Pi_i)$) of the transformation of network $\mathcal{N}$.

One transformation rule can match on many processes and one process can be matched on by many transformation rules. However, for the sake of simplicity, the transformation of $\mathcal{N}$ is split in several *transformation steps*. Since we assume that rule systems are confluent the order of transformation steps is irrelevant for the final result.

A transformation step transforms a network $\mathcal{N}$ of size n given a vector $\bar{M}$ consisting of n match pairs taken from $\mathfrak{m} \cup \{\delta\}$ where δ is a *dummy match pair* corresponding to a *dummy transformation rule* Δ that leaves the target process unchanged (i.e., $T(\Pi_i) = \Pi_i$ for some process LTS Π_i). The dummy transformation rule consists of a single state and no transitions in both its left and right patterns. The left and right matches of the i^{th} match pair $\bar{M}_i$ are referred to as m_i and $\hat{m}_i$, respectively. For each index $i \in 1..n$ the matches of a match pair $\bar{M}_i$ match on process LTS Π_i, i.e., we have $m_i : \mathcal{L}_i \to \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \to T(\Pi_i)$.

Each match pair in $(m_i, \hat{m}_i) \in \bar{M}$ corresponds to a rule $r \in R \cup \{\Delta\}$. Hence, the match pair vector $\bar{M}$ defines a partial mapping between processes in Π and rules in R. With abuse of notation, we shall use $\bar{M}$ as a partial mapping to project the synchronisation laws of Σ on Π according to the matches in $\bar{M}$. We write $\bar{M}(i) = j$ to indicate that m_i and $\hat{m}_i$ are matches for the j^{th} transformation rule of R. If $\bar{M}_i = \delta$, then we write $\bar{M}(i) = *$ to indicate that i is not mapped to a rule in R. This mapping describes a projection from synchronisation vectors of rule system Σ on to synchronisation vectors of network $\mathcal{N}$ on which the transformation step is applied. This *projection of synchronisation vectors* is formally defined as follows.

Definition 3.4.2 (Projection of synchronisation vectors)**.** *Let* $f : 1..n \nrightarrow 1..m$ *with* $n, m \in \mathbb{N}$ *be a partial mapping. For* $i \in 1..n$ *we write* $f(i) = *$ *to indicate that* i *is not mapped by* f*. A synchronisation vector* $\bar{v}$ *of size* m *can be projected iff for all* $j \in Ac(\bar{v})$

there exists an $i \in 1..n$ *such that* $f(i) = j$*. This condition ensures that all active indices of* $\bar{v}$ *are represented in the projected vector. The* projected synchronisation vector*, denoted* $\bar{v}^f$ *, is a vector of size* n *with elements* $i \in 1..n$ *defined as:*

$$\bar{v}_i^f = \begin{cases} \bullet & \text{if } f(i) = * \\ \bar{v}_{f(i)} & \text{otherwise} \end{cases}$$

Let f be a partial mapping. Given a synchronisation law $(\bar{v}, a)$ the *projected synchronisation law* is written as $(\bar{v}^f, a)$. The *projection of a set of synchronisation laws* $\mathcal{V}$ is defined as $\mathcal{V}^f = \{(\bar{v}^f, a) \mid (\bar{v}, a) \in \mathcal{V}\}$.

For a vector of matches $\bar{M}$ consisting of matches of $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ on an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$, the transformation step is formalised in Definition 3.4.3. The transformed LTS network $T_{\bar{M}}(\mathcal{N})$ consists of the transformed process LTSs and the original set of synchronisation laws $\mathcal{V}$ updated with the projection of $\hat{\mathcal{V}}$.

Before the transformation step defined by $\bar{M}$ can be applied it must be ensured that the input network $\mathcal{N} = (\Pi, \mathcal{V})$ contains the corresponding projection of the set of synchronisation laws $\mathcal{V}'$ of the applied rule system Σ, i.e., we must check $\mathcal{V}'^{\bar{M}} \subseteq \mathcal{V}$. If this is the case, then Σ is applicable, and the transformed network $T_{\bar{M}}(\mathcal{N})$ receives the set of synchronisation laws $\mathcal{V} \cup \hat{\mathcal{V}}^{\bar{M}}$. That is, the projected laws of the set of synchronisation laws $\hat{\mathcal{V}}$ are introduced to the network in the transformation step.

Each process LTS Π_i $(i \in 1..n)$ is transformed to an LTS $T(\Pi_i)$ by applying the matches $m_i : \mathcal{S}_{\mathcal{L}_i} \to \mathcal{S}_{\Pi_i}$ and $\hat{m}_i : \mathcal{S}_{\mathcal{R}_i} \to \mathcal{S}_{T(\Pi_i)}$. Note that every process LTS is matched on, since Δ matches on every state.

Definition 3.4.3 (LTS network transformation step)**.** *Let* $\mathcal{N} = (\Pi, \mathcal{V})$ *be an LTS network of size* n *and let* $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ *be a rule system. Let* $\bar{M}$ *be a vector of size* n *of match pairs* $(m_i, \hat{m}_i)$ *such that* $\mathcal{V}'^{\bar{M}} \subseteq \mathcal{V}$*. For all* $i \in 1..n$ *let* T_{m_i} *denote the LTS transformation of* Π_i *using the application of matches* m_i *and* $\hat{m}_i$ *according to Definition 3.3.4.*

The application of a transformation step *defined by* $\bar{M}$ *to LTS network* $\mathcal{N}$ *is defined as follows:*

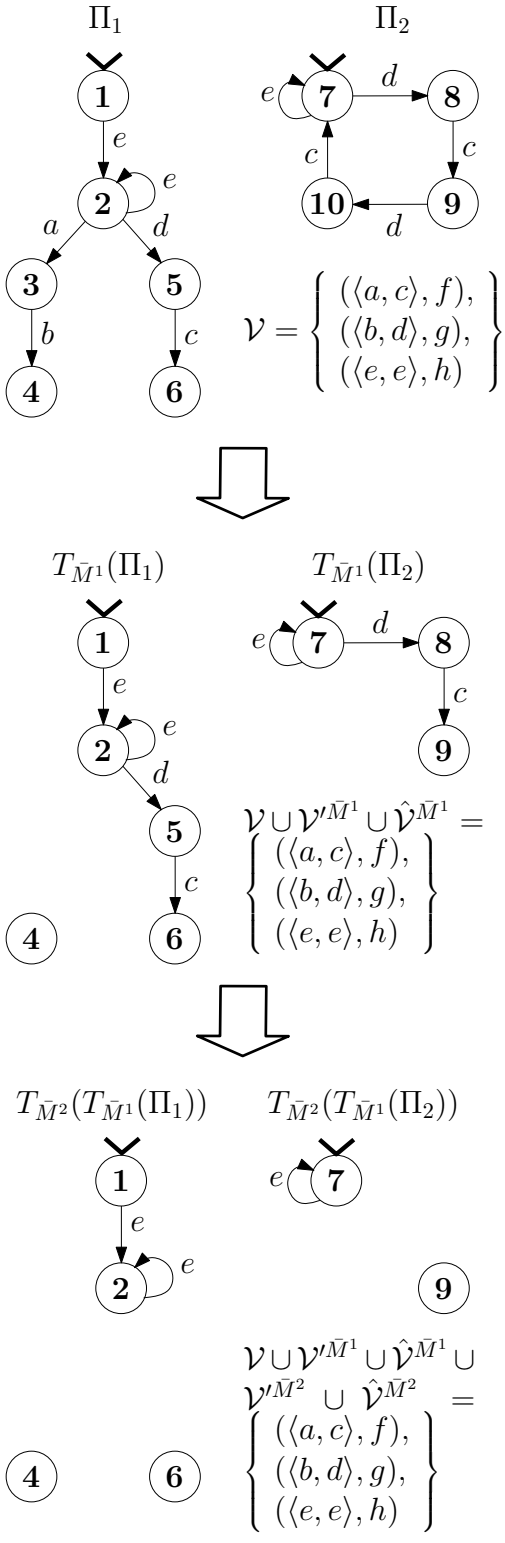
$$T_{\bar{M}}(\mathcal{N}) = (\langle T_{m_1}(\Pi_1), \ldots, T_{m_n}(\Pi_n)\rangle, \mathcal{V} \cup \hat{\mathcal{V}}^{\bar{M}})$$

The exhaustive application of a rule system Σ to a network $\mathcal{N}$ via a finite number of transformation steps is denoted by $T_\Sigma(\mathcal{N})$. The transformation proceeds via a series of match vectors $\bar{M}^1, \ldots, \bar{M}^n$, where n is the number transformation steps applied to obtain $T_\Sigma(\mathcal{N})$.

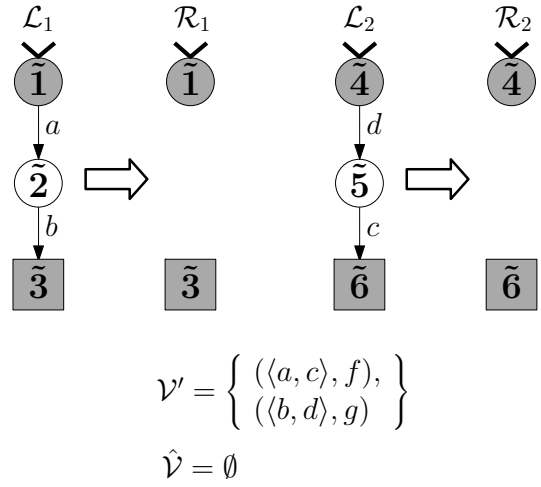
In most of our examples $\bar{M}$ is trivial and there is only a single transformation step. In these examples the definition of $\bar{M}$ is omitted and the transformed network is referred to as $T_\Sigma(\mathcal{N})$.

Figure 3.7 presents a transformation sequence that is the result of the application of a rule system Σ (see Figure 3.7b) to a network $\mathcal{N}$. The system LTSs of the input and output networks are exactly the same, this LTS is shown in Figure 3.7c. The a- and c-transitions and the b- and d-transitions can never synchronise. Furthermore, the d- and c- transitions in Π_1 are cut. The synchronisation of the e-transitions, resulting in h-transitions, are the only reachable behaviour in the system LTS.

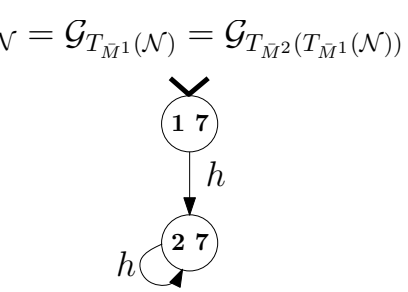
Rule system Σ removes the a-, b-transition sequence, and the d-, c-transition sequence. The system LTS $\mathcal{G}_\mathcal{N}$ of the network $\mathcal{N}$ that Σ is applied to remains unchanged, because in the described situation synchronisation of both the a- and d-transitions is impossible and



(a) A transformation sequence resulting from the application of Σ to $\mathcal{N} = (\Pi, \mathcal{V})$ with match pair vectors $\bar{M}^1$ and $\bar{M}^2$.



(b) Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ removes the a-, b-transition sequence, and the d-, c-transition sequence; the system LTS of networks Σ is applied to remain unchanged as the a- and d-transitions can both never synchronise and the b- and c-transitions are therefore unreachable.



(c) The system LTSs of the networks in the transformation sequence; since the a- and c-transitions and the b- and d-transitions can never synchronise and the d- and c-transitions in Π_1 are cut the h-transitions are the only reachable behaviour.

Figure 3.7: Exhaustive application of Σ to a network $\mathcal{N} = (\Pi, \mathcal{V})$ where $\bar{M}^1$ contains the left matches $m_1^1 : \mathcal{L}_1 \to \Pi_1$ and $m_2^1 : \mathcal{L}_2 \to \Pi_2$ with $m_1^1 = \{\tilde{1} \mapsto 2, \tilde{2} \mapsto 3, \tilde{3} \mapsto 4\}$ and $m_2^1 = \{\tilde{4} \mapsto 9, \tilde{5} \mapsto 10, \tilde{6} \mapsto 7\}$, and $\bar{M}^2$ contains the left matches $m_1^2 : \mathcal{L}_2 \to \Pi_1$ and $m_2^2 : \mathcal{L}_2 \to \Pi_2$ with $m_1^2 = \{\tilde{4} \mapsto 2, \tilde{5} \mapsto 5, \tilde{6} \mapsto 6\}$ and $m_2^2 = \{\tilde{4} \mapsto 7, \tilde{5} \mapsto 8, \tilde{6} \mapsto 9\}$

the b- and c-transitions are otherwise unreachable. Transformations like this are useful to gain insights in the reachable behaviour of local process LTSs.

Figure 3.7a shows the exhaustive application of Σ to a network $\mathcal{N} = (\Pi, \mathcal{V})$. The *first transformation step* applies the match pair vector $\bar{M}^1$ which contains the left matches $m_1^1 : \mathcal{L}_1 \to \Pi_1$ and $m_2^1 : \mathcal{L}_2 \to \Pi_2$ with $m_1^1 = \{\tilde{1} \mapsto 2, \tilde{2} \mapsto 3, \tilde{3} \mapsto 4\}$ and $m_2^1 = \{\tilde{4} \mapsto 9, \tilde{5} \mapsto 10, \tilde{6} \mapsto 7\}$. The projected set of synchronisation laws $\mathcal{V}'^{\bar{M}^1}$ is equivalent to $\mathcal{V}'$, i.e., $\mathcal{V}'^{\bar{M}^1} = \mathcal{V}'$. The resulting network is $T_{\bar{M}^1}(\mathcal{N}) = (\langle T_{m_1^1}(\Pi_1), T_{m_2^1}(\Pi_2)\rangle, \mathcal{V} \cup \mathcal{V}'^{\bar{M}^1} \cup \hat{\mathcal{V}}^{\bar{M}^1})$. The *second transformation step* applies the match pair vector $\bar{M}^2$ which contains the left matches $m_1^2 : \mathcal{L}_2 \to \Pi_1$ and $m_2^2 : \mathcal{L}_2 \to \Pi_2$ with $m_1^2 = \{\tilde{4} \mapsto 2, \tilde{5} \mapsto 5, \tilde{6} \mapsto 6\}$ and $m_2^2 = \{\tilde{4} \mapsto 7, \tilde{5} \mapsto 8, \tilde{6} \mapsto 9\}$. The projected set of synchronisation laws $\mathcal{V}'^{\bar{M}^2}$ is empty as for each $(\bar{v}, a) \in \mathcal{V}'$ we have $1 \in Ac(\bar{v})$ and there is no $i \in \{1, 2\}$ such that $\bar{M}(i) = 1$. This final transformed network is $T_\Sigma(\mathcal{N}) = (\langle T_{m_1^2}(T_{m_1^1}(\Pi_1)), T_{m_2^2}(T_{m_2^1}(\Pi_2))\rangle, \mathcal{V} \cup \mathcal{V}'^{\bar{M}^1} \cup \hat{\mathcal{V}}^{\bar{M}^1} \cup \mathcal{V}'^{\bar{M}^2} \cup \hat{\mathcal{V}}^{\bar{M}^2})$.

3.4.2 Analysing Transformations of an LTS Network

In a rule system, transformation rules can be dependent on each other regarding the behaviour they affect. In particular, the rules may refer to actions that require synchronisation according to some law, either in the network being transformed, or the network resulting from the transformation. Since in general, it is not known a priori whether or not those synchronisations can actually happen (see Figure 2.1, synchronisation of the c-transitions versus the synchronisation of the a- and e-transitions), full analysis of such rules must consider both successful and unsuccessful synchronisations.

To achieve this, dependent rules must be analysed together as combinations of LTS patterns, as shown in Figure 3.1. To this end, LTS patterns are combined into an LTS network, called a *pattern network* $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$, with $\bar{\Phi}$ a vector of pattern LTSs, and $\mathcal{W}$ a set of synchronisation laws. In particular, the left and right pattern networks of a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ are defined as $\bar{\mathcal{L}} = (\langle \mathcal{L}_1, \ldots, \mathcal{L}_{|R|}\rangle, \mathcal{V}')$ and $\bar{\mathcal{R}} = (\langle \mathcal{R}_1, \ldots, \mathcal{R}_{|R|}\rangle, \mathcal{V}' \cup \hat{\mathcal{V}})$. For the analysis of these pattern networks, we define in Definition 3.4.4 the *κ-extended pattern network* consisting of the combination of the κ-extended LTS patterns and an extension of the synchronisation laws with κ-synchronisation laws $\mathcal{V}^\kappa$. The left and right κ-extended pattern networks are denoted $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ and, for the purpose of equivalence checking, must use the same set of κ-synchronisation laws $\mathcal{V}^\kappa$.

Definition 3.4.4 (κ-Extended Pattern Network)**.** *Given a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n, its* κ-extended pattern network *is defined as $\mathcal{P}^\kappa$, where*

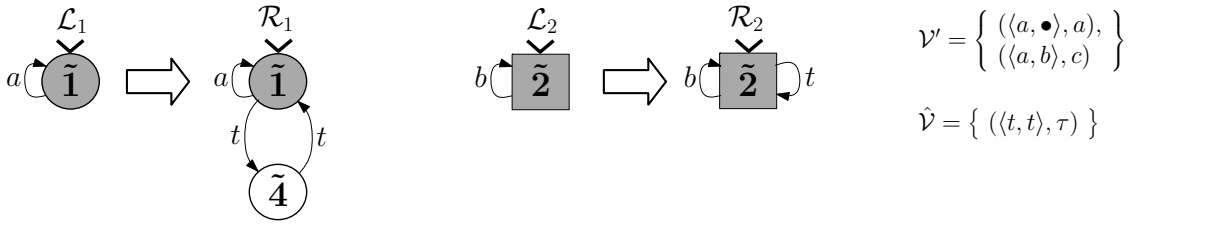
$$\mathcal{P}^\kappa = (\langle \bar{\Phi}_1^\kappa, \ldots, \bar{\Phi}_n^\kappa\rangle, \mathcal{W} \cup \mathcal{W}^\kappa), \text{ and}$$

$$\mathcal{W}^\kappa = \{(\bar{v}, \mu) \mid Ac(\bar{v}) \neq \emptyset \wedge \forall i \in Ac(\bar{v}). \\ ((\exists p \in \mathcal{I}_{\bar{\Phi}_i}.\ \bar{v}_i = \sigma_p) \vee (\exists p \in \mathcal{E}_{\bar{\Phi}_i}.\ \bar{v}_i = \varepsilon_p) \vee (\exists p \in \mathcal{E}_{\bar{\Phi}_i}, p' \in \mathcal{I}_{\bar{\Phi}_i}.\ \bar{v}_i = \gamma_{p,p'}))\}$$

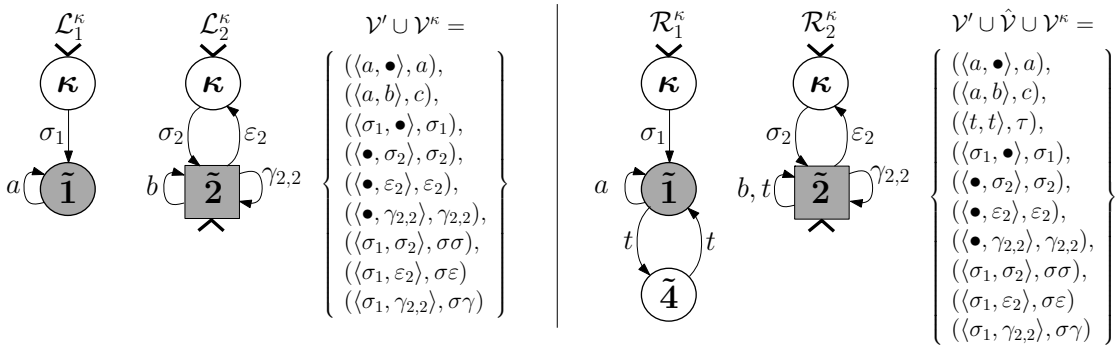
with μ an action that is unique with respect to rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$, i.e., $\mu \neq \tau \wedge \forall (\bar{v}', a) \in \mathcal{V}' \cup \hat{\mathcal{V}}.\ \mu \neq a$).

Verifying a rule system must account for all possible ways of entering or leaving the pattern networks. Therefore, the set of κ-synchronisation laws $\mathcal{W}^\kappa$ describes all possible combinations of synchronisations between σ-, ε-, and γ-actions.

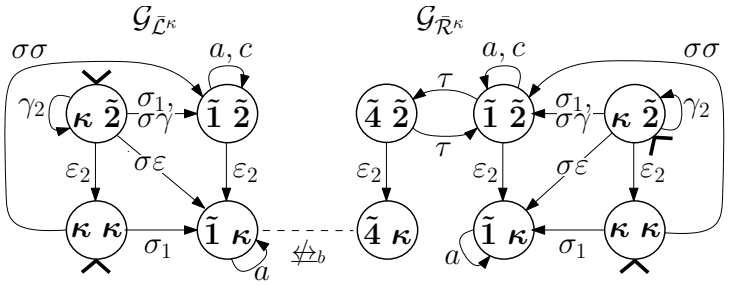
Figure 3.8 shows a rule system Σ, in which the two rules are dependent. Again, the states are numbered such that matches can be identified by the state label, i.e., a state $\tilde{i}$ is



(a) A rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$



(b) The corresponding κ-extended pattern networks



(c) Bisimulation check; the ε_2-transition ensures that $\bar{\mathcal{L}}^\kappa \not\leftrightarrows_b \bar{\mathcal{R}}^\kappa$

Figure 3.8: A rule system and its κ-extended pattern networks and bisimulation checks

matched onto state i. The two transformation rules depicted in Figure 3.8a introduce a new dependency between two (possibly) independent systems. The corresponding κ-extended pattern networks are given in Figure 3.8b. The κ-synchronisation laws allow σ-, ε-, and γ-actions to be performed both independently and synchronised. The synchronisations of σ_1- and σ_2-transitions, σ_1- and ε_2-transitions, and σ_1- and $\gamma_{2,2}$-transitions are displayed as the $\sigma\sigma$-transition, the $\sigma\varepsilon$-transition, and the $\sigma\gamma$-transition, respectively. Figure 3.8c presents the branching bisimulation check performed on the two κ-extended pattern networks. The check concludes that the two networks are not branching bisimilar. In particular, when the second process ($\mathcal{R}_2^\kappa$) leaves the pattern LTS at state $\langle \tilde{4}, \tilde{2}\rangle$ via the ε_2-transition, the a-transition can no longer be mimicked. The same would occur in any application of Σ at any state matched by $\langle \tilde{4}, \tilde{2}\rangle$ that has a transition to an unmatched state. [5]

[5] In our previous work [59], this type of rule system was called a non-cascading rule system. In this work, we no longer need to verify whether a rule system is cascading since the κ-state makes the effect of the transformation on the unmatched states explicit. Furthermore, the correctness proof of the technique presented in this chapter does not distinguish between cascading and non-cascading rule systems. Hence, the verification technique as described here will correctly reject non-cascading rule systems.

A rule system may consist of multiple classes of dependent rules where synchronisation is contained within a class. There is no synchronisation defined between the classes, i.e., the classes are independent of each other in terms of synchronising behaviour. These independent classes can be analysed separately.

Given a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$, the potential synchronisation between the behaviour in transformation rules in R is characterised by the *direct dependency relation* D:

$$D = \{(i,j) \mid \exists(\bar{v}, a) \in \mathcal{V}' \cup \hat{\mathcal{V}}.\ \{i,j\} \subseteq Ac(\bar{v})\}$$

Transformation rule R_i is related via D to the rule R_j iff both rules participate in a synchronisation law $(\bar{v}, a) \in \mathcal{V}' \cup \hat{\mathcal{V}}$. The relation considering directly and indirectly dependent rules, called the *dependency relation*, is defined by the transitive closure of D, i.e., D^+. The D^+ relation can be used to construct a partition $\mathbb{D}$ of transformation rule indices into classes of indices referring to dependent rules. Each class can be analysed independently. We call these classes *dependency sets*.

To define the projection of a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ along a dependency set $P \in \mathbb{D}$ we use $(P, <)$ to obtain a vector mapping the domain $1..|P|$ to the rules in R; we write $P(i) = j$ iff the i^{th} element of P (with $i \in 1..|P|$) refers to the transformation rule R_j (with $j \in 1..|R|$). The projection of a rule system along dependency set P is defined as follows.

Definition 3.4.5 (Projection of a rule system)**.** *Let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system with a partition $\mathbb{D}$ of dependency sets. The projection of Σ along a dependency set $P \in \mathbb{D}$ is a rule system $\Sigma^P = (R^P, \mathcal{V}'^P, \hat{\mathcal{V}}^P)$ with R^P a vector of size $|P|$ such that for all $i \in |P|$ we have $R^P_i = R_{P(i)}$.*

The left and right pattern networks of a projected rule system are denoted as $\bar{\mathcal{L}}^P$ and $\bar{\mathcal{R}}^P$.

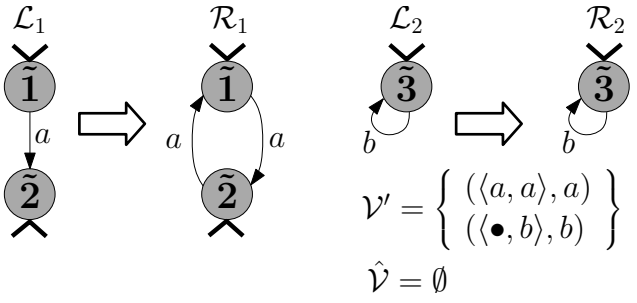
An analysis of the pattern networks is only sufficient if all relevant behaviour is described in those networks. Furthermore, the effect of the matches (i.e., the application of the rule system) must be taken into consideration with respect to both the projection of the sets of synchronisation laws $\mathcal{V}'$ and $\hat{\mathcal{V}}$, and completeness of transformation of synchronising transitions. To ensure the soundness of the transformation verification approach *one analysis condition* and *four application conditions* must be satisfied.

In Sections 3.4.2.1 and 3.4.2.2 the analysis of a rule system and application of a rule system, respectively, is discussed further. The analysis of a rule system consists of the verification of the pattern networks and the analysis condition. The analysis of the application of a rule system constitutes the verification of the application conditions. Both sections present an analysis algorithm and a time complexity analysis.

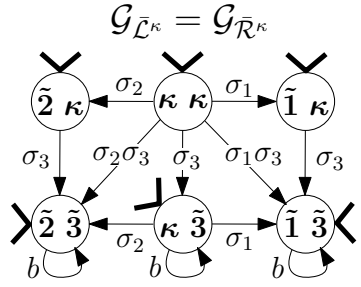
3.4.2.1 Analysis of a Rule System

In the analysis of a rule system the left and right pattern networks are checked for branching bisimilarity. To guarantee the soundness of this check, an analysis condition must apply. We first describe the analysis condition. Then, the algorithm for the analysis of a rule system is presented. Finally, this section is concluded with a run time analysis.

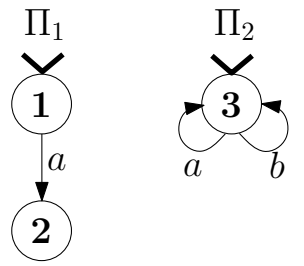
Consider a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$. The *analysis condition* requires that Σ is complete with respect to the synchronisation laws in $\mathcal{V}'$, That is, all the action labels described by the laws in $\mathcal{V}'$ must be transformed by the associated transformation rule. This ensures that any behaviour described in $\mathcal{V}'$, and affected by the rule system, is



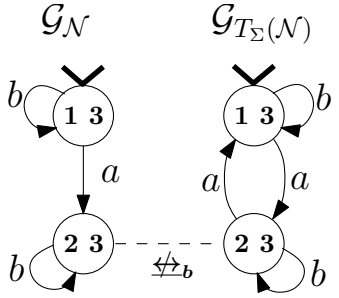
(a) A rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ that does not satisfy ANC1



(b) $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ are equivalent since a-transitions are not considered by rule 2



(c) An input network $\mathcal{N} = (\Pi, \mathcal{V})$



(d) The system LTSs before ($\mathcal{N}$) and after ($T_\Sigma(\mathcal{N})$) transformation are not branching bisimilar

Figure 3.9: Rule system Σ does not satisfy ANC1; although $\bar{\mathcal{L}}^\kappa \leftrightarrows_b \bar{\mathcal{R}}^\kappa$, a network $\mathcal{N}$ exists such that $\mathcal{G}_{\mathcal{N}} \not\leftrightarrows_b \mathcal{G}_{T_\Sigma(\mathcal{N})}$

explicitly visible in the pattern networks. The symmetric condition involving the $\mathcal{R}_i$ and $\hat{\mathcal{V}}$ applies as well.

$$\forall i \in 1..|R|.\ (\forall(\bar{v}, a) \in \mathcal{V}'.\ \bar{v}_i \in \mathcal{A}_{\mathcal{L}_i} \cup \{\bullet\}) \wedge (\forall(\bar{v}, a) \in \hat{\mathcal{V}}.\ \bar{v}_i \in \mathcal{A}_{\mathcal{R}_i} \cup \{\bullet\}) \qquad \text{(ANC1)}$$

Figure 3.9 shows how the application of a rule system that does not satisfy ANC1 affects the transformation verification. The rule system Σ, shown in Figure 3.9a, has a synchronisation law $(\langle a, a\rangle, a) \in \mathcal{V}'$. However, transformation rule R_2 does not contain any a-transitions, i.e., ANC1 is not satisfied. As a result the effects of the transformation of the a-transition by rule R_1 is not visible in the κ-extended pattern networks presented in Figure 3.9b. Figure 3.9d shows that an input network $\mathcal{N}$ exists (Figure 3.9c) such that the input network $\mathcal{N}$ and output network $T_\Sigma(\mathcal{N})$ are not branching bisimilar. If rule R_2 would contain the a-loop, then the a-transition would not have been cut and $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ would not be bisimilar any longer. Hence, it is vital that labels considered by synchronisation laws in $\mathcal{V}'$ are also present in the transformation rules, i.e., rule systems must adhere to ANC1.

The analysis In the *verification* of a *rule system* Σ the aim is to determine whether Σ is sound for any network $\mathcal{N}$ on which Σ is applicable. Before analysing the transformation rules with branching bisimulation checks, it is checked whether Σ is confluent and satisfies ANC1. *Verification of a rule system* $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ proceeds as follows:

1. Check whether in Σ, no τ-transitions can be synchronised, renamed, or cut, and whether ANC1 is satisfied. If not, report which check failed and stop.
2. Check whether the rules in Σ are confluent. If not, report that Σ is not confluent and stop.
3. For each rule in R the κ-extended pattern LTSs are constructed according to Definition 3.3.5.
4. Construct the set of dependency sets $\mathbb{D}$.
5. For each class (dependency set) $P \in \mathbb{D}$ determine whether $\bar{\mathcal{L}}^{\kappa,P} \underline{\leftrightarrow}_b \bar{\mathcal{R}}^{\kappa,P}$ holds, i.e., whether the κ-extended pattern networks projected along P are branching bisimilar.

If all steps produce positive results, then Σ is branching-structure preserving for all inputs it is applicable to. Otherwise, Σ may preserve the branching-structure of some LTS networks, but it certainly is not branching-structure preserving for *all possible* inputs it is applicable to.

Time complexity of the analysis In the *first* step of the verification of a rule system Σ, each check requires the verification of a condition on each synchronisation law in $\mathcal{V}'$, $\hat{\mathcal{V}}$, or both. Each condition can be checked in linear time. Hence, the running time of step 1 is $O(|\mathcal{V}' \cup \hat{\mathcal{V}}|)$.

In the *second* step, it is checked whether Σ is confluent. Confluence checking of transformations of LTSs has $O(\binom{|R|}{2} \cdot s^2 \cdot t \cdot \log(s))$ time complexity [215], with s and t the largest number of states and transitions in an LTS pattern of a rule in Σ, respectively.

In the *third* step, for each transformation rule R_i, the left and right κ-extended pattern LTSs are built, resulting in $\mathcal{L}_i^\kappa$ and $\mathcal{R}_i^\kappa$, respectively. The pattern LTSs must only be extended once. Therefore, the running time of step 3 has time complexity $O(|R| \cdot g)$, with g the largest number of glue-states appearing in an LTS pattern of a rule in Σ.

The *fourth* step constructs the dependency sets by analysing the synchronisation laws in $\mathcal{V}' \cup \hat{\mathcal{V}}$. This can be done in $O(|\mathcal{V}' \cup \hat{\mathcal{V}}|)$ time.

In the *fifth* and last step, for each dependency set $P \in \mathbb{D}$ the pattern networks $\bar{\mathcal{L}}^{\kappa,P}$ and $\bar{\mathcal{R}}^{\kappa,P}$ are constructed and it is verified whether $\bar{\mathcal{L}}^{\kappa,P}$ and $\bar{\mathcal{R}}^{\kappa,P}$ are branching bisimilar. Hence, $|\mathbb{D}|$ bisimulation checks are performed. Let s, t and a be the largest number of states, transitions and action labels, respectively, appearing in the κ-extended pattern networks of Σ. Branching bisimilarity checking can be performed in $O(t \cdot \log(s + a))$ [91]. Therefore, the time complexity of the final step of the analysis is $O(|\mathbb{D}| \cdot (t \cdot \log(s + a)))$.

The running time of steps 3-5 together therefore amounts to $O(|\mathbb{D}| \cdot (t \cdot \log(s + a)) + |\mathcal{V}' \cup \hat{\mathcal{V}}| + |R| \cdot g)$. In contrast with previous work, the analysis presented here only requires a single bisimulation check per dependency set $P \in \mathbb{D}$ (versus $2^{|P|} - 1$ in previous work [217]). This improvement is made possible by the new correctness proof presented in Section 3.4.3.

3.4.2.2 Analysis of the Application of a Rule System

The analysis presented in the previous section is not enough to guarantee the soundness of the transformation verification technique. There are four more conditions that need to be taken into account when the rule system is applied to an input LTS network. We first describe these four application conditions. Then, the algorithm for the analysis of the

application of a rule system is presented. Finally, this section is concluded with a run time analysis.

Consider a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ and an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of size n on which Σ is applied subject to a set of match pairs $\mathfrak{m}$.

The *first* condition concerns the completeness of transformation of synchronising transitions when applying rule system Σ to network $\mathcal{N}$. To prevent breaking branching bisimilarity due to a mixture of old and new synchronising behaviour, we require that old synchronising behaviour is completely transformed. A rule transforming synchronising transitions (with a minimum of two synchronising parties) must be applicable to all equivalent synchronising transitions. More precisely, for each active action label $\bar{v}_j$ ($j \in |R|$) of a law $(\bar{v}, a) \in \mathcal{V}'$ that synchronises with another action label (i.e., $\{j\} \subset Ac(\bar{v})$), we must have that if a process Π_i ($i \in 1..n$) is matched on by $\mathcal{L}_j$, *all* $\bar{v}_j$-transitions in Π_i are transformed, i.e., for all $\bar{v}_j$-transitions in Π_i, there exists a match pair $(m, \hat{m})$ such that $m{:}\mathcal{L}_j \to \Pi_i$ matches a $\bar{v}_j$-transition in $\mathcal{L}_j$ on that $\bar{v}_j$-transition in Π_i.

$$\begin{aligned}
&\forall j \in 1..|R|, (\bar{v}, a) \in \mathcal{V}'.\ \{j\} \subset Ac(\bar{v}) \wedge \bar{v}_j \in \mathcal{A}_{\mathcal{L}_j} \implies \\
&\quad \forall i \in 1..n, (s, \bar{v}_j, s') {\in} \mathcal{T}_i.\ \exists (m{:}\mathcal{L}_j \to \Pi_i, _) \in \mathfrak{m}, (p, \bar{v}_j, p') {\in} \mathcal{T}_{\mathcal{L}_j}. \\
&\qquad m(p) = s \wedge m(p') {=} s'
\end{aligned} \tag{APC1}$$

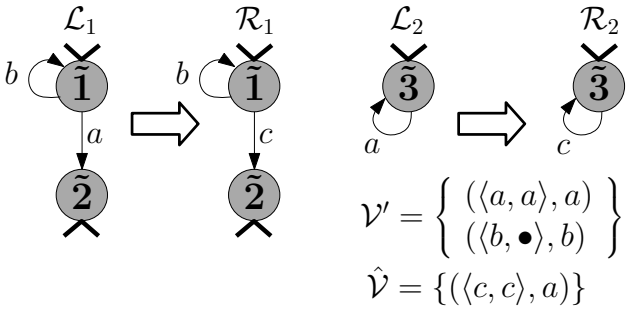
We write "_" to indicate that the second element of the match pair is not relevant. The symmetric condition involving the $\mathcal{R}_j$ and $\mathcal{V}' \cup \hat{\mathcal{V}}$ applies as well. Together with ANC1, APC1 ensures that synchronising transitions with a particular label in the input network are either all transformed, or none are transformed. This is shown in Section 3.4.3 in Lemma 3.4.15.

Figure 3.10 shows a transformation that satisfies ANC1, but does not adhere to APC1. The rule system Σ, presented in Figure 3.10a, transforms a-transitions to c-transitions. The first transformation rule transforms an a-transition to a c-transition iff there is a b-loop at the state from which the a-transition is performed. The second transformation rule transforms a-loops to c-loops. If Σ is applied to network $\mathcal{N}$, presented in Figure 3.10c, then the transition $\langle 1 \rangle \xrightarrow{a}_{\Pi_1} \langle 2 \rangle$ is not transformed. Therefore, the transformation does not satisfy APC1.

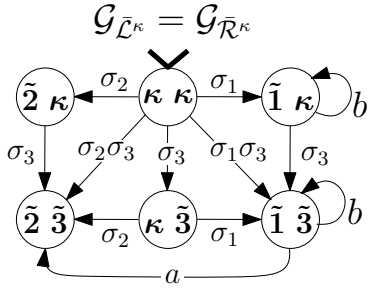
The laws of the rule system describe that the synchronisation of two a-transitions results in an a-transition (i.e., $(\langle a, a \rangle, a) \in \mathcal{V}'$), and that the b-loop is performed independently of other processes (i.e., $(\langle b, \bullet \rangle, b) \in \mathcal{V}'$). A new synchronisation law $(\langle c, c \rangle, a)$ is added such that the synchronisation of two c-transitions results in an a-transition again. This makes old and new synchronising behaviour comparable. As shown in Figure 3.10b, the branching bisimulation check cannot distinguish between the left and right κ-extended pattern networks.

However, if Σ is applied to the LTS network $\mathcal{N}$ given in Figure 3.10c, then it turns out that $\mathcal{G}_{\mathcal{N}}$ and $\mathcal{G}_{T_\Sigma(\mathcal{N})}$ are not branching bisimilar (see Figure 3.10d). The transformed network can no longer perform the $\langle 2, 3 \rangle \xrightarrow{a} \langle 1, 3 \rangle$ transition. Transition $\langle 2 \rangle \xrightarrow{a} \langle 1 \rangle$ in process Π_1 has not been transformed while the a-loop in Π_2 has been transformed to a c-loop in $T(\Pi_2)$. Hence, there is no a-transition available anymore with which the $\langle 2 \rangle \xrightarrow{a}_{T(\Pi_2)} \langle 1 \rangle$ transition can synchronise.

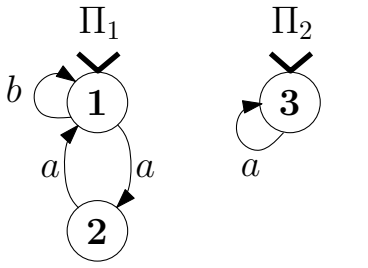
The *second* condition prevents that projections of new synchronisation laws in $\hat{\mathcal{V}}$ are defined over actions already present in the processes of an input network. Otherwise, an LTS network could be altered without actually defining any transformation rules. Formally, if the left LTS pattern $\mathcal{L}_j$ ($j \in |R|$) of the j^{th} rule in R is matched on the i^{th}



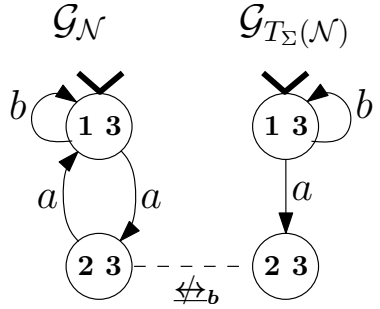
(a) Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ transforms a-transitions to c-transitions, synchronisation of c-transitions results in an a-transition



(b) $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ are equivalent since the synchronisation of two c-transitions result in an a-transition again



(c) An input network $\mathcal{N} = (\Pi, \mathcal{V})$



(d) The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; the transformed model is no longer able to synchronise the a-transition performed at state $\langle 2\rangle$ that was not transformed, since the loop at state $\langle 3\rangle$ has been transformed to a c-loop

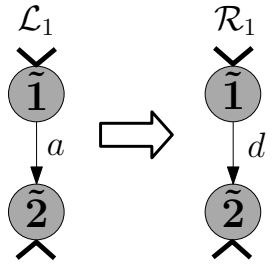
Figure 3.10: Rule system Σ and input network $\mathcal{N}$ with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy APC1; although $\bar{\mathcal{L}}^\kappa \underline{\leftrightarrow}_b \bar{\mathcal{R}}^\kappa$, the system LTS of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to the system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$

process ($i \in 1..n$) in Π, then the $\bar{v}_j$ may not be defined over actions in $\mathcal{A}_i$.

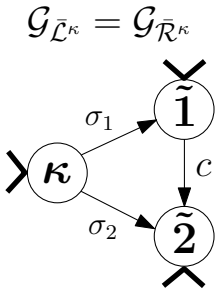
$$\forall i \in 1..n, j \in 1..|R|, (m{:}\mathcal{L}_j \to \Pi_i, \_) \in \mathfrak{m}, (\bar{v}, a) \in \hat{\mathcal{V}}, .\ \bar{v}_j \notin \mathcal{A}_i \qquad \text{(APC2)}$$

As an example, consider a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ with $\mathcal{V}' = \emptyset$, and $\hat{\mathcal{V}} = \{\langle a\rangle, b\}$. Say R contains a single transformation rule that transforms an a-loop with $\mathcal{L}_1 = \mathcal{R}_1$. Note that rule R_1 does not change the LTSs it is applied to, and thus $\bar{\mathcal{L}}^\kappa = \bar{\mathcal{R}}^\kappa$. Furthermore, APC1 and ANC1 are satisfied for both $\mathcal{V}'$ and $\hat{\mathcal{V}}$. If Σ is applied to a network $\mathcal{N} = (\Pi, \{(\langle a\rangle, a)\})$ with $\Pi = \langle \mathcal{L}_1 \rangle$, then we obtain the network $T_\Sigma(\mathcal{N}) = (\Pi, \{(\langle a\rangle, a), (\langle a\rangle, b))$. Clearly, the system LTSs of $\mathcal{N}$ and $T_\Sigma(\mathcal{N})$ are not branching bisimilar; $T_\Sigma(\mathcal{N})$ can perform both an a-loop and a b-loop whereas $\mathcal{N}$ can only perform the a-loop. Condition APC2 does not allow the application of Σ to $\mathcal{N}$ as $(\langle a\rangle, b) \in \hat{\mathcal{V}}$ involves label a which is present in $\mathcal{A}_1$.

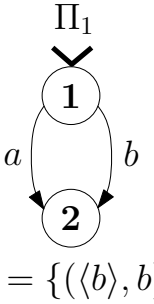
The *third* and *fourth* conditions concern how the set of laws $\mathcal{V}$ (of the network $\mathcal{N}$ to which Σ is applied) is related to the set of laws $\mathcal{V}'$ (that Σ expects). Consider a set of match pairs $\mathfrak{m}$ describing the transformation of $\mathcal{N}$ as defined by Σ. The application of the matches in $\mathfrak{m}$ is distributed over a sequence of transformation steps. Let $\bar{M}$ be a vector of match pairs defining a single transformation step in the sequence. For each



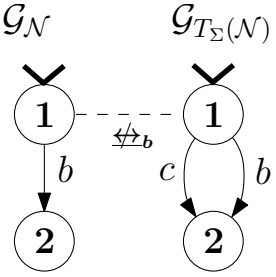
(a) Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ transforms a-transitions to d-transitions; the synchronisation laws specify that both the local a- and the d-transitions result in a global c-transition.



(b) $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ are equivalent since for both the left and right patterns the process-local transitions result in a c-transition.



(c) Input network $\mathcal{N} = (\Pi, \mathcal{V})$.



(d) The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; since $(\langle a \rangle, c) \in \mathcal{V}' \setminus \mathcal{V}$ (i.e., APC3 is violated) the a-transition is cut in Π_1 while the d-transition is not cut in $T(\Pi_1)$ due to introduction of $(\langle d \rangle, c)$.

Figure 3.11: Rule system Σ and input network $\mathcal{N}$ with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy APC3; although $\bar{\mathcal{L}}^\kappa \underline{\leftrightarrow}_b \bar{\mathcal{R}}^\kappa$, the system LTS of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to the system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$.

transformation step $\bar{M}$ it is required that APC3 and APC4 hold.

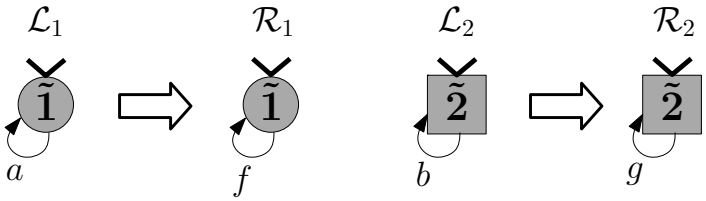
The *third* condition expresses that the set of synchronisation laws $\mathcal{V}$ of network $\mathcal{N}$ must contain all the synchronisation laws in $\mathcal{V}'$ that Σ expects.

$$\mathcal{V}'^{\bar{M}} \subseteq \mathcal{V} \tag{APC3}$$

An application of a rule system Σ to an LTS network $\mathcal{N}$ for which APC3 does not hold is given in Figure 3.11. The corresponding match pair vector is $\bar{M} = (m : \mathcal{L}_1 \to \Pi_1, \hat{m} : \mathcal{R}_1 \to T(\Pi_1))$. Condition APC3 does not hold since the law $(\langle a \rangle, c) \in \mathcal{V}'$ of rule system Σ, presented in Figure 3.11a is not included in the set of laws of the input network $\mathcal{N}$, shown in Figure 3.11c. The analysis condition ANC1 and application conditions APC1 and APC2 hold.

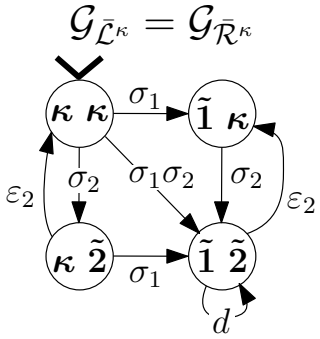
The rule system transforms a-transitions to d-transitions. The local a-transitions result in global c-transitions due to law $(\langle a \rangle, c) \in \mathcal{V}'$. To ensure that the behaviour remains equivalent a new synchronisation law $(\langle d \rangle, c) \in \hat{\mathcal{V}}$ is introduced such that, like the a-transitions, the d-transitions result in global c-transitions. As shown in Figure 3.11b the left and right κ-extended pattern networks are branching bisimilar, as expected.

However, when Σ is applied to input network $\mathcal{N}$ the transformation of the a-transition in process Π_1 (now a d-transition) is not cut due to introduction of the law $(\langle d \rangle, c) \in \hat{\mathcal{V}}$.

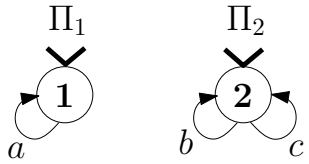


$$\mathcal{V}' = \{(\langle a, b\rangle, d)\} \qquad \hat{\mathcal{V}} = \{(\langle f, g\rangle, d)\}$$

(a) Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ transforms a-loops to f-loops and b-loops to g-loops; like synchronisation of a and b labels, the synchronisation of f and g labels results in a d-transition.

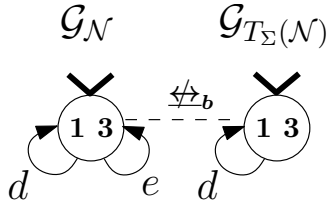


(b) $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ are equivalent since the transformed loops synchronise to a d-loop again.



$$\mathcal{V} = \left\{ \begin{array}{l} (\langle a, b\rangle, d) \\ (\langle a, c\rangle, e) \end{array} \right\}$$

(c) Input network $\mathcal{N} = (\Pi, \mathcal{V})$.



(d) The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; the transformed model is unable to synchronise the b-transition of Π_2 that was not transformed because the a-loop in Π_1 has been transformed to an f-loop.

Figure 3.12: Rule system Σ and input network $\mathcal{N}$ with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy APC4; although $\bar{\mathcal{L}}^\kappa \leftrightarrow_b \bar{\mathcal{R}}^\kappa$, the system LTS of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to the system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$

The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are given in Figure 3.11d. Since $\mathcal{V}'^{\bar{M}} \not\subseteq \mathcal{V}$, an analysis of Σ does not take into account that in Π_1 the a-transition is cut. Therefore, the analysis cannot give any guarantees for the input network $\mathcal{N}$.

If application condition APC3 is satisfied by a network, then either network $\mathcal{N}$ must include the law $(\langle a\rangle, c) \in \mathcal{V}$ or the law must be removed from $\mathcal{V}'$ in rule system Σ. In the former case, the a-transition in Π_1 is not cut and the system LTS $\mathcal{G}_{\mathcal{N}}$ and $\mathcal{G}_{T_\Sigma(\mathcal{N})}$ are branching bisimilar. In the latter case, the a-transition in $\mathcal{L}_1$ is cut as well and it follows that $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ are not branching bisimilar. Condition APC3 ensures that, with respect to the transitions described in the transformation rules, both the rules system and the input network cut the same transitions.

The *fourth* condition ensures that Σ is aware of all the synchronisation laws in $\mathcal{V}$ that affect the rules in R. That is, besides the projection of synchronisation laws in $\mathcal{V}'$, no other synchronisation laws in $\mathcal{V}$ may involve behaviour described by the rules in R.

$$\forall(\bar{v}, a) \in \mathcal{V} \setminus \mathcal{V}'^{\bar{M}}, i \in Ac(\bar{v}).\ \bar{v}_i \notin \mathcal{A}_{\mathcal{L}_{\bar{M}(i)}} \tag{APC4}$$

The symmetric condition involving the $\mathcal{V} \setminus \hat{\mathcal{V}}$ and $\mathcal{A}_{R_{\bar{M}(i)}}$ applies as well.

A transformation that does not satisfy APC4 is presented in Figure 3.12. Condition APC4 is not satisfied because the law $(\langle a, c\rangle, e) \in \mathcal{V} \setminus \mathcal{V}'$ of input network $\mathcal{N}$, shown in

Figure 3.12c, contains behaviour that influences the transformation rules of rule system Σ, shown in Figure 3.12a. The transformation satisfies conditions ANC1, APC1, APC2, and APC3.

Rule system Σ transforms a-loops to f-loops and b-loops to g-loops. In an attempt to preserve the semantics the f- and g-actions, like the a- and b-actions, are forced to synchronise, resulting in d-actions. As a result the left and right κ-extended pattern networks, presented in Figure 3.12b, are branching bisimilar.

However, if Σ is applied to network $\mathcal{N}$, then the possibility of synchronising the a- and c-loops is lost. It follows that $\mathcal{G}_{\mathcal{N}}$ can perform an e-loop while $\mathcal{G}_{T_\Sigma(\mathcal{N})}$ cannot (see Figure 3.12d). Hence, the two system LTSs are not branching bisimilar.

If APC4 is satisfied, then rule system Σ must contain the synchronisation law $(\langle a, c\rangle, e) \in \mathcal{V}'$. Additionally, due to ANC1, the b-transition must then be present in $\mathcal{L}_2$. It then becomes visible when comparing the κ-extended pattern networks $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ that the possibility of performing an e-loop is lost in $\bar{\mathcal{R}}^\kappa$.

Note that for a confluent rule system all transformation sequences have the same end result. Therefore, it is sufficient that these conditions hold for a single transformation sequence from the input network to the final output network.

The analysis For the *application* of a rule system Σ the aim is to determine whether a verified Σ is sound for a network $\mathcal{N}$ to which Σ is applied. Before transformation of a network $\mathcal{N}$, it is checked whether the application conditions APC1, APC2, APC3, APC4 are satisfied. Checking *applicability* of a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ to an input network $\mathcal{N} = (\Pi, \mathcal{V})$ is performed as follows:

1. Calculate the maximum set of match pairs $\mathfrak{m}$.
2. Check whether APC1 and APC2 hold for all $(m, \hat{m}) \in \mathfrak{m}$.
3. Distribute the match pairs in $\mathfrak{m}$ over a sequence of transformation steps defined by $\bar{M}^1, \ldots, \bar{M}^k$ with $k \in \mathbb{N}$.
4. Check whether APC3 and APC4 are satisfied with respect to each $\bar{M}^i$ $(i \in 1..k)$.

If the steps return positive results, then Σ is applicable to $\mathcal{N}$.

Time complexity of the analysis To check for the applicability of a rule system Σ, a set of matches is required. Say that n is the size of the input network $\mathcal{N}$, m is the size of the set of matches, and t is the largest number of transitions in an LTS pattern of a rule in Σ.

In the *first* step, the maximum set of matches is calculated. In general, the graph matching problem [68] is NP-complete. However, Dodds and Plum [68] have shown that if target graph $\mathcal{G}$ has a root (the initial state), the left pattern $\mathcal{L}$ has a set of unique labelled root states (the glue-states), all states are reachable from these roots, and each state has a bounded number b of outgoing transitions, then the complexity is independent of the size of the input graph, instead only depending on b and the number of transitions n in $\mathcal{L}$. The complexity is then $O(\Sigma_{i=0}^{n} b_i)$. The reachability assumption is easily satisfied for finite systems as unreachable states can be removed from process LTSs and LTS patterns. Moreover, for systems with a finite number of action labels the boundedness of the out-degree of states is also satisfied.

In the *second* analysis step, application conditions APC1 and APC2 are verified. When APC1 is checked, for each law $(\bar{v}, a) \in \mathcal{V}'$ with $|\bar{v}| > 1$, it is checked whether all $\bar{v}_i$-transitions ($i \in 1..n$) are matched. At worst this takes $O(n \cdot m \cdot t \cdot |\mathcal{V}'|)$. When APC2 is checked, for each match and each law $(\bar{v}, a) \in \hat{\mathcal{V}}$ it is checked whether for all $i \in 1..n$ it holds that $\bar{v}_i$ is not an action of the corresponding matched process. This takes $O(n \cdot m \cdot |\hat{\mathcal{V}}|)$ time. Hence, the running time of the second step is $O(n \cdot m \cdot (|\hat{\mathcal{V}}| + t \cdot |\mathcal{V}'|))$.

The *third* step distributes the matches in $\mathfrak{m}$ over a sequence of match pairs. The set is traversed as a vector of match pairs of size n that contains as many pairs from $\mathfrak{m}$ as possible. The time complexity is $O(m^2)$.

The *fourth* and final step verifies whether APC3 and APC4 hold. The check for both APC3 and APC4 needs to iterate over vectors of match pairs $\mathcal{N}^i$ ($i \in 1..k$) and indices of all synchronisation laws in $\mathcal{V}$ and $\mathcal{V}'$. The number of matches in the match vector is limited to the number of matches in the set $\mathfrak{m}$. Therefore, this step has a running time of $O(n \cdot m \cdot |\mathcal{V}| \cdot |\mathcal{V}'|)$.

The running time of steps 2-4 together therefore amounts to $O(n \cdot m \cdot (|\mathcal{V}| \cdot |\mathcal{V}'| + |\hat{\mathcal{V}}| + t \cdot |\mathcal{V}'|))$. If we assume that $|\hat{\mathcal{V}}| \leq |\mathcal{V}|$ and $|\mathcal{V}'| \leq |\mathcal{V}|$, then the running time simplifies to $O(n \cdot m \cdot |\mathcal{V}| \cdot (|\mathcal{V}| + t + 1))$.

3.4.3 Correctness of the verification

In this section, we prove the correctness of the rule system verification as described in the previous section. We prove the soundness of the rule system verification in Proposition 3.4.16. The completeness of the verification approach is shown in Proposition 3.4.17.

To simplify the proofs, we strengthen condition APC1 such that the correctness proof can be formulated on a single transformation step instead of a sequence. Application condition APC1 is formulated over a set of matches. However, since rule systems are confluent there is always a single result LTS after a series of applications of a rule system. Therefore, we may consider a 'merged' match without influencing the correctness of the verification technique over confluent sequences.

In line with this simplification, we assume that Σ has n rules, and that a rule R_i ($i \in 1..n$) matches on Π_i in the LTS network that Σ is applied to. For a single transformation step, the rules in R can be reordered according to this assumption with an appropriate projection of the rule system. For confluent rule systems, the result can be lifted to confluent sequences of transformations steps and the strengthened APC1 can be *weakened* again to APC1.

In this case, where R_i matches on Π_i, we do not have to consider the projection of synchronisation laws since $\mathcal{V} = \mathcal{V}^{\bar{M}}$ for a set of synchronisation laws $\mathcal{V}$ and vector of match pairs $\bar{M}$. Hence, we simply *write* $\mathcal{V}$ *instead of* $\mathcal{V}^{\bar{M}}$.

To prove soundness of the technique, we show that from a bisimulation relation B between $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$, a bisimulation relation C can be constructed between an arbitrary $\mathcal{N}$ and corresponding $T_\Sigma(\mathcal{N})$. For this purpose, we first need to prove some lemmas. For clarity, we refer with $\bar{q}, \hat{\bar{q}}, \bar{q}', \ldots$ to states in a left pattern network, with $\bar{p}, \hat{\bar{p}}, \bar{p}', \ldots$ to states in a right pattern network, with $\bar{s}, \hat{\bar{s}}, \bar{s}', \ldots$ to states in an input network, and with $\bar{t}, \hat{\bar{t}}, \bar{t}', \ldots$ to states in an output network.

The κ-extended pattern networks can be seen as an abstraction from the input networks. In a κ-extended pattern network, individual processes can only leave the κ-state via σ-transitions and only enter the κ-state via ε-transitions. Hence, for all transitions enabled by laws in the original (non-κ-extended) pattern network, the processes that are in the

κ-state before such a transition is executed are still in the κ-state after the transition has been followed. This property is formalised in Lemmas 3.4.6 and 3.4.7.

Lemma 3.4.6. *Consider a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n. Then,*

$$\forall \bar{v}' \in \mathcal{W}, \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}' \implies \forall i \in 1..n.\ \bar{p}_i = \kappa \iff \bar{p}'_i = \kappa$$

Proof. Let $(\bar{v}, a) \in \mathcal{W}$ and $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ such that $\bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}'$. Let $i \in 1..n$. We distinguish two cases: $\bar{p}_i = \kappa$ or $\bar{p}'_i = \kappa$. We only discuss the first case, the proof for the other case is symmetric.

Say $\bar{p}_i = \kappa$. By Definition 3.3.5, σ-transitions can only be performed from a κ-state. Similarly, a process can only enter a κ-state by performing an ε-transition. Hence, since $(\bar{v}, a) \in \mathcal{W}$ and $\bar{p}_i = \kappa$, we must have $\bar{v}_i = \bullet$. It follows from Definition 2.3.3 that $\bar{p}_i = \bar{p}'_i$. Hence, we have $\bar{p}'_i = \kappa$.

Since the proof for case $\bar{p}'_i = \kappa$ is symmetric, it follows that $\bar{p}_i = \kappa \iff \bar{p}'_i = \kappa$. □

Lemma 3.4.6 can be applied inductively to obtain a similar result for τ-paths. Synchronisation laws with τ as a result action are, by Definition 3.4.4, never κ-synchronisation laws. Therefore, every process that is in a κ-state before a sequence of τ-transitions is still in the κ-state after the sequence of τ-transitions as shown by Lemma 3.4.7.

Lemma 3.4.7. *Consider a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n. Then,*

$$\forall \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}^\kappa} \bar{p}' \implies \forall i \in 1..n.\ \bar{p}_i = \kappa \iff \bar{p}'_i = \kappa$$

Proof. Consider states $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ such that $\bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}^\kappa} \bar{p}'$. The use of τ-actions is not allowed in laws in $\mathcal{W}^\kappa$, hence it follows that $(\bar{v}, a) \in \mathcal{W}$. Therefore, we have $\bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}} \bar{p}'$. The remainder of the proof follows directly from Lemma 3.4.6 and structural induction on $\bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}} \bar{p}'$. □

Due to the κ-laws, branching bisimulation relations between $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ preserve κ-states, i.e., when two state vectors are related, any κ-states present in one vector are also present in the other vector at the same positions, and vice versa. This is expressed in Lemma 3.4.8.

Lemma 3.4.8. *Consider two pattern networks $\bar{\mathcal{L}}$ and $\bar{\mathcal{R}}$ of size n. Then,*

$$\forall \bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^\kappa}, \bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}.\ \bar{p} \leftrightarrows_b \bar{q} \implies \forall i \in 1..n.\ \bar{p}_i = \kappa \iff \bar{q}_i = \kappa$$

Proof. Consider states $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^\kappa}$ and $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$. For each $i \in 1..n$, we can distinguish two symmetric cases: $\bar{p}_i = \kappa$ or $\bar{q}_i = \kappa$. We only discuss the first case, the proof for the other case is symmetric.

Say $\bar{p}_i = \kappa$. By Definition 2.2.1, there is at least one state $\hat{p} \in \mathcal{I}_{\mathcal{L}_i}$, and furthermore, according to Definition 3.3.5, there is a transition $\kappa \xrightarrow{\sigma_{\hat{p}}}_{\mathcal{L}_i} \hat{p}$. Hence, there is a law $(\bar{v}, \mu) \in \mathcal{V}^\kappa$, with $\bar{v}_i = \sigma_{\hat{p}}$ and $\forall j \in 1..n.\ j \neq i \Rightarrow \bar{v}_j = \bullet$, enabling transition $\bar{p} \xrightarrow{\mu}_{\bar{\mathcal{L}}^\kappa} \bar{p}'$ for some $\bar{p}'$ with $\bar{p}'_i = \hat{p}$ (by Definition 3.4.4). Since $\bar{p} \leftrightarrows_b \bar{q}$ and $\mu \neq \tau$, we have $\bar{q} \xrightarrow{\tau}{}^* \hat{\bar{q}} \xrightarrow{\mu} \bar{q}'$. It follows that $\hat{\bar{q}}_i \xrightarrow{\sigma_{\hat{p}}} \bar{q}'_i$. Since σ-transitions can only be performed from κ-states, we have $\hat{\bar{q}}_i = \kappa$. Finally, from Lemma 3.4.7 it follows that $\bar{q}_i = \kappa$.

Since the proof for case $\bar{q}_i = \kappa$ is symmetric, we have $\forall i \in 1..n.\ \bar{p}_i = \kappa \iff \bar{q}_i = \kappa$. □

As κ-extended pattern networks form an abstraction from the matched input network, it is desirable that those states representing states not removed by the transformation are related to themselves. In the κ-extended left and right pattern networks the glue-states and the κ-states represent the states that are kept. As shown in Lemma 3.4.9, this can be directly lifted to the network-global level when state vectors only contain exit- and κ-states. However, this cannot be guaranteed for state vectors that also contain in-states due to the lack of a unique transition (such as the σ-, ε, and γ-transitions) leaving those in-states. If a state vector $\bar{p}$ consists of in-, out- and κ-states, then $\bar{p}$ may be related to a different state vector $\bar{q}$ via a τ-path originating from an in-state $\bar{p}_i$ contained in $\bar{p}$. When matches on initial states are restricted to exit-states, Lemma 3.4.9 is sufficient to show that initial states of the input and output networks of a transformation are related.

Lemma 3.4.9. *Consider a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ and a branching bisimulation relation B such that $\bar{\mathcal{L}}^\kappa \; B \; \bar{\mathcal{R}}^\kappa$. Then, $\forall \bar{p} \in \mathcal{I}_{\bar{\mathcal{L}}^\kappa}.\ \bar{p} \; B \; \bar{p}$.*

Proof. Consider a state $\bar{p} \in \mathcal{I}_{\bar{\mathcal{L}}^\kappa}$. We will construct a state $\bar{p}'$ and synchronisation law $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$ such that $\bar{p} \xrightarrow{\bar{v}^\kappa, \mu} \bar{p}'$. We construct $\bar{v}^\kappa$ and $\bar{p}'$ with for all $i \in 1..|R|$:

$$(\bar{v}_i^\kappa, \bar{p}_i') = \begin{cases} (\varepsilon_{\bar{p}_i}, \kappa) & \text{if } \bar{p}_i \in \mathcal{E}_{\bar{\Phi}_i} \\ (\sigma_x, x) & \text{if } \bar{p}_i = \kappa. \text{ By Definition 2.2.1, there exists an } x \in \mathcal{I}_{\bar{\mathcal{L}}_i} \end{cases}$$

Let μ be the unique result action corresponding to $\bar{v}^\kappa$. Since for all $i \in 1..|R|$ either $\bar{p}_i = \kappa$ or $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i}$, there is a transition $\bar{p}_i \xrightarrow{\bar{v}_i^\kappa} \bar{p}_i'$. It follows that there is a transition $\bar{p} \xrightarrow{\bar{v}^\kappa, \mu} \bar{p}'$.

By Definition 2.2.2, there is a state $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$ such that $\bar{p} \; B \; \bar{q}$. Furthermore, since $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$, we have $\mu \neq \tau$. Hence, there is a $\bar{q} \xrightarrow{\tau}{}^* \hat{\bar{q}} \xrightarrow{\mu} \bar{q}'$ such that $\bar{p} \; B \; \hat{\bar{q}}$ and $\bar{p}' \; B \; \bar{q}'$. We show that $\bar{p} = \hat{\bar{q}}$ from which it follows that $\bar{p} \; B \; \bar{p}$. Consider an $i \in 1..n$. The σ-transitions only leave from κ-states (in which case $\bar{p}_i = \kappa$) and the $\varepsilon_{\bar{p}_i}$-transitions only leave from the state $\bar{p}_i$, i.e., each of the $\bar{p}_i \xrightarrow{\bar{v}_i^\kappa} \bar{p}_i'$ transitions carries a unique label identifying the states connected by the transition. Both $\bar{p}_i$ and $\hat{\bar{q}}_i$ can perform the $\xrightarrow{\bar{v}_i^\kappa}$ directly. It follows that $\bar{p}_i = \hat{\bar{q}}_i$. Therefore, $\bar{p} = \hat{\bar{q}}$ and $\bar{p} \; B \; \hat{\bar{q}}$ can be rewritten to $\bar{p} \; B \; \bar{p}$. □

To formally define how a κ-extended network relates to an input network, we introduce a mapping of state vectors as presented in Definition 3.4.10. Similar to matches for a single rule, the mapping of a state vector of a κ-extended pattern network defines how a state vector of the pattern is mapped to a state vector of an LTS network.

Definition 3.4.10 (State vector mapping)**.** *Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with corresponding matches $m_i : \bar{\Phi}_i \to \Pi_i$ for all $i \in 1..n$. We say a state vector $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$ is* mapped *to a state vector $\bar{s} \in \mathcal{S}_\mathcal{N}$, denoted by $\bar{p} \vdash \bar{s}$, iff*

$$\forall i \in 1..n. \begin{pmatrix} (\bar{p}_i \neq \kappa \implies m_i(\bar{p}_i) = \bar{s}_i) \\ \wedge\ (\bar{p}_i = \kappa \implies \neg \exists x \in \mathcal{S}_{\bar{\Phi}_i}, m_i(x) = \bar{s}_i) \end{pmatrix}$$

By referring to matches of the individual vector elements, a state vector is mapped on to another state vector. Since the κ-state represents unmatched states, the mapping relates the κ-state to all unmatched states. Hence, for every state $\bar{s} \in \mathcal{S}_\mathcal{N}$ there is a state $\bar{p} \in \mathcal{P}^\kappa$ that maps on state $\bar{s}$ (Lemma 3.4.11).

Since κ-states represent all unmatched states, we need to construct states that specify explicitly which unmatched state is represented at the moment. The state $\bar{s}' := \bar{s}[\bar{p}_i \mid P(i)]$ denotes the state $\bar{s}'$ constructed from states $\bar{s}$ and $\bar{p}$ such that for all $i \in 1..n$, if predicate $P(i)$ holds, we have $\bar{s}'_i = \bar{p}_i$, and if not, we have $\bar{s}'_i = \bar{s}_i$. For example, the state $\bar{s}' := \bar{s}[m_i(\bar{p}_i) \mid \bar{p}_i \neq \kappa]$ is produced from matches of $\bar{p}$, where for all $i \in 1..n$, $\bar{s}'_i = m_i(\bar{p}_i)$ if $\bar{p}_i \neq \kappa$, and $\bar{s}'_i = \bar{s}_i$ if $\bar{p}_i = \kappa$.

With the exception of transitions enabled by $\mathcal{W}^\kappa$, the input network is able to simulate the behaviour of the κ-extended network. The transitions enabled by κ-laws form an abstraction from all transitions that may possibly enter or leave states of the input network matched by the glue-states of the pattern network. That is, for laws $(\bar{v}, a) \in \mathcal{W}$, the mapping preserves the branching structure of the pattern network. Following, we formalise this in a number of lemmas.

The state vector mapping preserves the branching structure of the pattern network Similar to a match, the state vector mapping (Definition 3.4.10) preserves the branching structure of the pattern network for the set of matching laws. Before proving this claim, we first show that the state vector mapping is complete, i.e., the mapping relation maps to all states of any input network. More precisely, for each (vector) state $\bar{s}$ in the input network there is a (vector) state in the κ-extended pattern network that is mapped on $\bar{s}$. This is formally proven in Lemma 3.4.11.

Lemma 3.4.11. *Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ and corresponding matches $m_i : \bar{\Phi}_i \to \Pi_i$ for all $i \in 1..n$. Then, $\forall \bar{s} \in \mathcal{S}_\mathcal{N}.\ \exists \bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{p} \vdash \bar{s}$.*

Proof. Let $\bar{s}$ be a state in $\mathcal{S}_\mathcal{N}$. From the definition of state vector mapping (Definition 3.4.10) it follows that there is a $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$ with $\bar{p} \vdash \bar{s}$. We shall construct $\bar{p}$ such that $\bar{p} \vdash \bar{s}$. Consider an $i \in 1..n$. If $\exists x \in \mathcal{S}_{\bar{\Phi}_i}.\ m_i(x) = \bar{s}_i$, then we take $\bar{p}_i = x$. Otherwise, we take $\bar{p}_i = \kappa$. By construction, it holds that $\bar{p} \vdash \bar{s}$. Finally, by Definition 3.3.5, $x, \kappa \in \mathcal{S}_{\bar{\Phi}^\kappa_i}$, and thus, $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$. □

Lemmas 3.4.12 and 3.4.13 express that the state vector mapping preserves the branching structure of the pattern network. Lemma 3.4.12 states that for each transition in the pattern network there is a corresponding transition in the mapped network. Lemma 3.4.13 extends this to sequences of τ-transitions. In Lemma 3.4.12 and Lemma 3.4.13, the end state of the transition and the sequence of transitions in the input network, respectively, is identified. In short, for vector states $\bar{p}$ and $\bar{s}$, if $\bar{p} \vdash \bar{s}$, then for each index i, firstly, transitions taken in pattern $\bar{\Phi}_i$ can be mimicked by transitions in process Π_i from the mapped state leading to a state mapped by the target state in $\bar{\Phi}_i$, and secondly, if the state $\bar{p}_i$ is a κ-state, then no transitions from the corresponding state $\bar{s}_i$ are taken.

Lemma 3.4.12. *Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ (APC3) and corresponding matches $m_i : \bar{\Phi}_i \to \Pi_i$ for all $i \in 1..n$. Then,*

$$\forall (\bar{v}, a) \in \mathcal{W}, \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}' \implies$$
$$\forall \bar{s} \in \mathcal{S}_\mathcal{N}.\ \bar{p} \vdash \bar{s} \implies \exists \bar{s}' \in \mathcal{S}_\mathcal{N}.\ \bar{p}' \vdash \bar{s}' \wedge \bar{s} \xrightarrow{\bar{v},a}_\mathcal{N} \bar{s}' \wedge \forall i \in 1..n.\ \bar{p}'_i = \kappa \implies \bar{s}'_i = \bar{s}_i$$

Proof. Consider synchronisation law $(\bar{v}, a) \in \mathcal{W}$ and states $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ such that $\bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}'$, and $\bar{s} \in \mathcal{S}_\mathcal{N}$ with $\bar{p} \vdash \bar{s}$. Take $\bar{s}' := \bar{s}[m_i(\bar{p}'_i) \mid \bar{p}'_i \neq \kappa]$. By construction, we have $\bar{s}' \in \mathcal{S}_\mathcal{N}$ and $\forall i \in 1..n.\ \bar{p}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{s}_i$. Furthermore, Lemma 3.4.6 ensures that $\bar{p}_i = \kappa \iff \bar{p}'_i = \kappa$ for all $i \in 1..n$. Therefore, for all $i \in 1..n$, it holds that $(\neg\exists x.m_i(x) = \bar{s}_i) \iff (\neg\exists x.m_i(x) = \bar{s}'_i)$. It follows that $\bar{p}' \vdash \bar{s}'$.

What is left to show is $(\bar{v}, a) \in \mathcal{V}$ and $\forall i \in 1..n.\ (\bar{v}_i = \bullet \implies \bar{s}_i = \bar{s}'_i \wedge \bar{s}_i \in \mathcal{S}_i) \wedge (\bar{v}_i \neq \bullet \implies \bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i)$ (by Definition 2.3.3). Since $\mathcal{W} \subseteq \mathcal{V}$ (by APC3) and $(\bar{v}, a) \in \mathcal{W}$, we must have $(\bar{v}, a) \in \mathcal{V}$. Consider an $i \in 1..n$. We distinguish two cases:

- $\bar{v}_i = \bullet$. As $\bar{s} \in \mathcal{S}_\mathcal{N}$, it holds that $\bar{s}_i \in \mathcal{S}_i$. Moreover, by Definition 2.3.3, we have $\bar{p}_i = \bar{p}'_i$ and $\bar{p}_i \in \mathcal{S}_{\bar{\Phi}^\kappa_i}$. If $\bar{p}_i = \kappa$, then by construction of $\bar{s}'$ we have $\bar{s}_i = \bar{s}'_i$. If $\bar{p}_i \neq \kappa$, then $m_i(\bar{p}_i) = \bar{s}_i$ and $m_i(\bar{p}'_i) = \bar{s}'_i$. Finally, since $\bar{p}_i = \bar{p}'_i$ and m_i is injective (Definition 3.3.3), $m_i(\bar{p}_i) = m_i(\bar{p}'_i) = \bar{s}_i = \bar{s}'_i$.

- $\bar{v}_i \neq \bullet$. By Definition 2.3.3 and $(\bar{v}, a) \in \mathcal{W}$, we have $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i} \bar{p}'_i$. Hence, it follows that $m_i(\bar{p}_i) = \bar{s}_i$ and $m_i(\bar{p}'_i) = \bar{s}'_i$. Finally, since a match (Definition 3.3.3) is an embedding, we conclude that $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$.

In conclusion, we have $(\bar{v}, a) \in \mathcal{V}$ and $\forall i \in 1..n.\ (\bar{v}_i = \bullet \implies \bar{s}_i = \bar{s}'_i \wedge \bar{s}_i \in \mathcal{S}_i) \wedge (\bar{v}_i \neq \bullet \implies \bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i)$. Therefore, it holds that $\bar{s} \xrightarrow{\bar{v},a}_\mathcal{N} \bar{s}'$. □

Lemma 3.4.13. *Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ (APC3) and corresponding matches $m_i : \bar{\Phi}_i \to \Pi_i$ for all $i \in 1..n$. Then,*

$$\forall \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}^\kappa} \bar{p}' \implies$$
$$\forall \bar{s} \in \mathcal{S}_\mathcal{N}.\ \bar{p} \vdash \bar{s} \implies (\exists \bar{s}' \in \mathcal{S}_\mathcal{N}.\ \bar{p}' \vdash \bar{s}' \wedge \bar{s} \xrightarrow{\tau}{}^*_\mathcal{N} \bar{s}' \wedge \forall i \in 1..n.\ \bar{p}'_i = \kappa \implies \bar{s}'_i = \bar{s}_i)$$

Proof. Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ and $\bar{s}, \bar{s}' \in \mathcal{S}_\mathcal{N}$ such that $\bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}^\kappa} \bar{p}'$, $\bar{p} \vdash \bar{s}$, and $\bar{p}' \vdash \bar{s}'$. Since τ result actions are not allowed in $\mathcal{W}^\kappa$, any law $(\bar{v}, \tau)$ in the sequence of τ's must be a member of $\mathcal{W}$. The proof now follows directly from Lemma 3.4.12 and structural induction on $\bar{p} \xrightarrow{\tau}{}^*_{\mathcal{P}^\kappa} \bar{p}'$. □

When two states $\bar{s}$ and $\bar{t}$ from the input and transformed network, respectively, are related by a branching bisimulation relation, and from $\bar{s}$ a certain transition is enabled, then from $\bar{t}$ it must be possible to simulate this behaviour (and vice versa). Even when this transition is not matched by the transformation rule system, it may still be the case that $\bar{t}$ is matched by a state that is not a glue-state. In this case, $\bar{t}$ cannot simulate $\bar{s}$ directly, thus, there must be a τ-path to some other state that is able to directly simulate $\bar{s}$. The existence of such a state is proven in Lemma 3.4.14.

In Lemma 3.4.14, we show that in the presence of a state vector mapping and a witness showing that there is a transition leaving the pattern, all *active* glue-states (all glue-states involved in that transition) or κ-states are reachable by related states. More precisely, when a state $\bar{p}$ – mapped to a state from which the pattern is left – is related to a state $\bar{q}$, then there is a τ-path from $\bar{q}$ to $\hat{\bar{q}}$ such that $\bar{p} \leftrightarrow_b \hat{\bar{q}}$ and there is a state $\bar{q}'$ which is the corresponding entry-point of a given unmatched transition that leaves $\bar{q}$. This follows from two facts. First, the κ-synchronisation laws have synchronisation vectors uniquely identifying the active states within $\bar{q}$. Second, due to the matching conditions (Definition 3.3.3), a matched state must be a glue-state when there is a transition leaving or entering the corresponding matched state.

Lemma 3.4.14. *Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network of size n, let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system applicable to $\mathcal{N}$, and let B be a branching bisimulation relation such that $\bar{\mathcal{L}}^\kappa \; B \; \bar{\mathcal{R}}^\kappa$. Let $\bar{M}$ be a vector of match pairs of size n such that $m_i : \mathcal{L}_i \to \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \to T(\Pi_i)$ for all $i \in 1..n$. Then,*

$$\begin{aligned}
&\forall (\bar{v}, a) \in \mathcal{V}, \bar{s}, \bar{s}' \in \mathcal{S}_\mathcal{N}.\ \bar{s} \xrightarrow{\bar{v},a}_\mathcal{N} \bar{s}' \implies \\
&\quad \forall \bar{p}, \bar{p}' \in \mathcal{S}_{\bar{\mathcal{L}}^\kappa}, \bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}.\ \bar{p}\ B\ \bar{q} \wedge \bar{p} \vdash \bar{s} \wedge \bar{p}' \vdash \bar{s}' \wedge (\forall i \in Ac(\bar{v}).\ \neg \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i} \bar{p}'_i) \implies \\
&\quad\quad \exists \hat{\bar{q}}, \bar{q}' \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}.\ \bar{q} \xrightarrow{\tau}{}^*_{\bar{\mathcal{R}}^\kappa} \hat{\bar{q}} \wedge \bar{p}\ B\ \hat{\bar{q}} \wedge \bar{p}'\ B\ \bar{q}' \wedge \\
&\quad\quad\quad (\forall i \in Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i) \wedge (\forall i \in 1..n \setminus Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{q}'_i)
\end{aligned}$$

Proof. Consider a synchronisation law $(\bar{v}, a) \in \mathcal{V}$ and states $\bar{s}, \bar{s}' \in \mathcal{S}_\mathcal{N}$ such that $\bar{s} \xrightarrow{\bar{v},a} \bar{s}'$. Let $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^\kappa}$ and $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$ be states such that $\bar{p}\ B\ \bar{q}$, $\bar{p} \vdash \bar{s}$, and $\forall i \in Ac(\bar{v}).\ \neg \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i} \bar{p}'_i$ (there is no transition in $\mathcal{T}_{\bar{\mathcal{L}}}$ matching $\bar{s} \xrightarrow{\bar{v},a}_\mathcal{N} \bar{s}'$).

A κ-extended pattern network explicitly models transitions that enter and leave the embedding of the pattern network. However, it does not model the situation where the matched network moves between two unmatched states. Therefore, we have to perform a case distinction: either the transition of the input-network $\mathcal{N}$ is represented by one of the κ-synchronisations, or the transition of the input network has no representation in $\bar{\mathcal{L}}^\kappa$. In the former case, we build the corresponding κ-synchronisation law $(\bar{v}^\kappa, \mu)$ and obtain the required states by applying the branching bisimulation definition to $\bar{p}\ B\ \bar{q}$ and $\bar{p} \xrightarrow{\bar{v}^\kappa,\mu} \bar{p}'$. In the latter case, we show that $\bar{p} = \bar{p}'$ and take $\bar{q}$ for both $\hat{\bar{q}}$ and $\bar{q}'$ from which the proof will follow.

We distinguish the two aforementioned cases:

C1 There exists an $i \in Ac(\bar{v})$ such that $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$, $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa)$, or $(\bar{p}_i = \kappa \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$. The three cases correspond to the situations where the γ-, ε-, and σ-transitions, respectively, are introduced in κ-extended pattern networks. We will construct a synchronisation law $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$ enabling a transition $\bar{p} \xrightarrow{\mu}_{\bar{\mathcal{L}}^\kappa} \bar{p}'$ that represents the LTS pattern network abstraction for $\bar{s} \xrightarrow{\bar{v},a}_\mathcal{N} \bar{s}'$. We construct $\bar{v}^\kappa$ with for all $i \in 1..n$:

$$\bar{v}^\kappa_i = \begin{cases} \gamma_{\bar{p}_i, \bar{p}'_i} & \text{if } i \in Ac(\bar{v}) \wedge \bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \\ \varepsilon_{\bar{p}_i} & \text{if } i \in Ac(\bar{v}) \wedge \bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa \\ \sigma_{\bar{p}'_i} & \text{if } i \in Ac(\bar{v}) \wedge \bar{p}_i = \kappa \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \\ \bullet & \text{otherwise} \end{cases}$$

Let μ be the unique result action corresponding to $\bar{v}^\kappa$. Since there are no matching transitions $(\forall i \in Ac(\bar{v}).\ \neg \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i} \bar{p}'_i)$, by Definition 3.3.3, for all $i \in Ac(\bar{v})$ we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. It follows that $(\bar{v}^\kappa, \mu)$ indeed enables the transition $\bar{p} \xrightarrow{\mu}_{\bar{\mathcal{L}}^\kappa} \bar{p}'$. Furthermore, by Definition 3.4.4, we have $\mu \neq \tau$. Since $\bar{p}\ B\ \bar{q}$ and $\mu \neq \tau$, by Definition 2.2.2, we have $\bar{q} \xrightarrow{\tau}{}^*_{\bar{\mathcal{R}}^\kappa} \hat{\bar{q}} \xrightarrow{\mu}_{\bar{\mathcal{R}}^\kappa} \bar{q}'$ with $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$. What remains to be shown is 1) $\forall i \in Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i$ and 2) $\forall i \in 1..n \setminus Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{q}'_i$:

1) Consider an $i \in Ac(\bar{v})$. We distinguish two cases:

- $i \in Ac(\bar{v}^\kappa)$. Because μ is unique in $\mathcal{V}^\kappa$ and does not occur in $\mathcal{V}' \cup \hat{\mathcal{V}}$, the transition $\hat{\bar{q}} \xrightarrow{\mu}_{\bar{\mathcal{R}}^\kappa} \bar{q}'$ is enabled by $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$. Recall that $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$ (since $i \in Ac(\bar{v})$). The $\bar{v}^\kappa_i$-transition is only present between $\bar{p}_i$ and $\bar{p}'_i$ in both $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$, and (by Definition 3.3.5). Therefore, we must have $\hat{\bar{q}}_i = \bar{p}_i$ and $\bar{q}'_i = \bar{p}'_i$.
- $i \notin Ac(\bar{v}^\kappa)$. It follows that $\bar{p}_i = \bar{p}'_i$ and $\hat{\bar{q}}_i = \bar{q}'_i$ (Definition 2.3.3). Recall that $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$ (since $i \in Ac(\bar{v})$). However, since $i \notin Ac(\bar{v}^\kappa)$ only the case where both $\bar{p}_i$ and $\bar{p}'_i$ are κ-states remains. By applying Lemma 3.4.8 to $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$, it follows that $\hat{\bar{q}}_i = \kappa$ and $\bar{q}'_i = \kappa$. Hence, $\bar{p}_i = \bar{p}'_i$ and $\hat{\bar{q}}_i = \bar{q}'_i$.

2) Consider an $i \in 1..n \setminus Ac(\bar{v})$. Since $i \notin Ac(\bar{v})$, we must have $i \notin Ac(\bar{v}^\kappa)$ (by construction of $\bar{v}^\kappa$). It follows from Definition 2.3.3 that $\hat{\bar{q}}_i = \bar{q}'_i$.

C2 For all $i \in Ac(\bar{v})$ it holds that $\neg(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$, $\neg(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa)$, and $\neg(\bar{p}_i = \kappa \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$.

We take $\bar{q}$ for both $\hat{\bar{q}}$ and $\bar{q}'$ which leads to $\bar{p}\ B\ \hat{\bar{q}}$. We first show that $\bar{p} = \bar{p}'$ from which it follows that $\bar{p}'\ B\ \bar{q}'$. The proof is then completed by showing $\forall i \in Ac(\bar{v}).\ \bar{q}_i = \bar{p}_i$, the remainder follows from $\hat{\bar{q}} = \bar{q}' = \bar{q}$.

We show that $\bar{p} = \bar{p}'$. Consider an $i \in 1..n$. If both $\bar{p}_i = \kappa$ and $\bar{p}'_i = \kappa$, we trivially have $\bar{p}_i = \bar{p}'_i$. Now consider the opposite case: $\bar{p}_i \neq \kappa$ or $\bar{p}'_i \neq \kappa$. Assume for a contradiction that $i \in Ac(\bar{v})$. Then by Definition 3.3.3, we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. Since $\bar{p}_i \neq \kappa$ or $\bar{p}'_i \neq \kappa$, only the following three cases remain: $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$, $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa)$, and $(\bar{p}_i = \kappa \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$. These three cases contradict the assumptions of case C2. Hence, we must have $i \notin Ac(\bar{v})$. It now follows that $\bar{s}_i = \bar{s}'_i$. Finally, we have $\bar{p}_i = \bar{p}'_i$ because m is an injection. In conclusion, we have $\bar{p} = \bar{p}'$.

What remains to be shown is $\forall i \in Ac(\bar{v}).\ \bar{q}_i = \bar{p}_i$. Consider an $i \in Ac(\bar{v})$. Again, by Definition 3.3.3, we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. Based on this we distinguish three cases: $\bar{p}_i = \kappa$, $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa$, and $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i}$. In the first case it follows from $\bar{p}\ B\ \bar{q}$ and Lemma 3.4.8 that $\bar{p}_i = \kappa = \bar{q}_i$. The latter two cases contradict one of the three assumptions of case C2 and the proof follows by contradiction.

In both C1 and C2 there exist $\hat{\bar{q}}, \bar{q}' \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$ such that $\bar{q} \xrightarrow{\tau}{}^*_{\bar{\mathcal{R}}^\kappa} \hat{\bar{q}}$ with $\bar{p}\ B\ \hat{\bar{q}}$, $\bar{p}'\ B\ \bar{q}'$, $\forall i \in Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i$, and $\forall i \in 1..n \setminus Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{q}'_i$. □

Completeness of transition transformation Due to the application conditions either all process-local transitions participating in a synchronising global transition are transformed or no process-local transitions are transformed at all. For confluent rule systems, the order of applying rules is irrelevant and application of the transformation rules can lead to only one output network. This fact allows us to simplify the correctness proof of the verification technique by strengthening condition APC1 such that the correctness proof (Proposition 3.3.9) can be formulated on a single transformation step instead of a sequence.

Recall that APC1 requires that a rule transforming synchronising transitions labelled with some action a must be applicable to all a-transitions within the corresponding LTS. Since confluent rule systems have only a single possible output network, we may consider a 'merged' match consisting of all the matches in the sequence that produces the output

network in a single transformation step. A transformation sequence resulting in the final output network cannot be distinguished from the application of the 'merged' match. Therefore, for proving the correctness of the technique, APC1 may be strengthened as follows:

$$\begin{aligned}&\forall \Pi_i \in \Pi, r_j \in R, (\bar{v}, a) \in \mathcal{V}'.\ \{j\} \subset Ac(\bar{v}) \wedge \bar{v}_j \in \mathcal{A}_{\mathcal{L}_j} \implies \\ &\quad \forall m{:}\mathcal{L}_j \to \Pi_i, (s, \bar{v}_j, s') {\in} \mathcal{T}_i.\ \exists (p, \bar{v}_j, p') {\in} \mathcal{T}_{\mathcal{L}_j}.\ m(p) = s \wedge m(p') {=} s'\end{aligned} \tag{APC1'}$$

In contrast to APC1, requiring existence of a match that transforms synchronising transitions for all equivalent transitions, the strengthened condition APC1' requires that a single match transforms *all* equivalent synchronising transitions. Indeed, this means that APC1' requires a 'merged' match transforming all synchronising transitions in one step.

Conditions APC1' and ANC1 ensure that global transitions in the input network are always fully transformed or not transformed at all. This leads to Lemma 3.4.15 that states the following: if a transition in a network enabled by $(\bar{v}, a) \in \mathcal{V}$ has a match on a local transition (say $\bar{v}_i$ for some $i \in 1..n$), then for all $j \in 1..n$, the participating local $\bar{v}_j$-transitions must be matched, i.e., all local transitions participating in the global transition must be matched. From this it follows that a global transition is either transformed fully or not transformed at all.

Lemma 3.4.15. *Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of size n and a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ such that APC1' and ANC1 are satisfied. Consider the pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ as representative for the left and right pattern network. Let the $m_i : \bar{\Phi}_i \to \Pi_i$ ($i \in 1..n$) be the matches specifying the embedding of $\mathcal{P}$ in $\mathcal{N}$. Then,*

$$\begin{aligned}&\forall (\bar{v}, a) \in \mathcal{W}, \bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{N}}, \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}.\ \bar{s} \xrightarrow{\bar{v},a}_{\mathcal{N}} \bar{s}' \wedge \bar{p} \vdash \bar{s} \wedge \bar{p}' \vdash \bar{s}' \wedge \\ &\quad (\exists j \in Ac(\bar{v}).\ m_j(\bar{p}_j) = \bar{s}_j \wedge m_j(\bar{p}'_j) = \bar{s}'_j \wedge \bar{p}_j \xrightarrow{\bar{v}_j}_{\bar{\Phi}_j} \bar{p}'_j) \implies \bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}'\end{aligned}$$

Proof. Consider a synchronisation law $(\bar{v}, a) \in \mathcal{W}$ and states $\bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{N}}$ such that $\bar{s} \xrightarrow{\bar{v},a}_{\mathcal{N}} \bar{s}'$. Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ with $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$. Finally, let there be an $j \in Ac(\bar{v})$ such that $m_j(\bar{p}_j) = \bar{s}_j$, $m_j(\bar{p}'_j) = \bar{s}'_j$, and $\bar{p}_j \xrightarrow{\bar{v}_j}_{\bar{\Phi}_j} \bar{p}'_j$ matches transition $\bar{s}_j \xrightarrow{\bar{v}_j}_j \bar{s}'_j$. We shall show that $\bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}'$ by showing that for all $i \in 1..n$ there is a transition $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i^\kappa} \bar{p}'_i$ if $\bar{v}_i \neq \bullet$, and $\bar{p}_i = \bar{p}'_i$ with $\bar{p}_i \in \mathcal{S}_{\bar{\Phi}_i^\kappa}$ if $\bar{v}_i = \bullet$ (Definition 2.3.3). Consider an $i \in 1..n$. We distinguish three cases:

- $\bar{v}_i \neq \bullet \wedge Ac(\bar{v}) = \{i\}$. Law $(\bar{v}, a)$ constitutes independent behaviour and the proof follows from the premises.
- $\bar{v}_i \neq \bullet \wedge \{i\} \subset Ac(\bar{v})$. Law $(\bar{v}, a)$ constitutes synchronising behaviour. Next we show that $\bar{v}_i \in \mathcal{A}_{\bar{\Phi}_i}$, after which we can apply APC1' and show that $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i^\kappa} \bar{p}'_i$. Since ANC1 is not symmetrical with respect $\bar{\mathcal{L}}$ and $\bar{\mathcal{R}}$ we need to distinguish these two cases showing $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$ and $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$ respectively.
  - $\mathcal{P} = \bar{\mathcal{L}}$. The left pattern network has the laws $\mathcal{W} = \mathcal{V}'$. Because $(\bar{v}, a) \in \mathcal{V}'$, by ANC1, we have $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$.
  - $\mathcal{P} = \bar{\mathcal{R}}$. The right pattern network has the laws $\mathcal{W} = \mathcal{V}' \cup \hat{\mathcal{V}}$, hence, $(\bar{v}, a) \in \mathcal{V}' \cup \hat{\mathcal{V}}$. In the trivial case, where $\bar{v} \in \hat{\mathcal{V}}$, it directly follows from ANC1 that

$\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$. In the other case, where $\bar{v} \in \mathcal{V}'$, Since $i \in Ac(\bar{v})$, there is a transition $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$ in the transformed network. This transition either originates from the network Σ is applied to, or is introduced by the transformation as specified in Definition 3.3.4. In the former case, we arrive at a contradiction: by the left variant of APC1' and ANC1, it follows that the original transition *is* matched on, while the transition is *not* matched on according to Definition 3.3.4. In the latter case, there exists $x, x' \in \mathcal{S}_{\mathcal{R}_i}$ with $x \xrightarrow{\bar{v}_i}_{\mathcal{R}_i} x'$. Therefore, we have $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$.

In all cases we have $\bar{v}_i \in \mathcal{A}_{\bar{\Phi}_i}$. We can now apply APC1' to obtain states $p, p' \in \mathcal{S}_{\bar{\Phi}_i}$ such that $p \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i} p'$. Since m_i is injective, we have $\bar{p}_i = p$ and $\bar{p}'_i = p'$. Hence, there is a transition $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}^\kappa_i} \bar{p}'_i$.

- $\bar{v}_i = \bullet$. By Definition 2.3.3, $\bar{s}_i = \bar{s}'_i$. Hence, since matches are injective it follows from $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$ that $\bar{p}_i = \bar{p}'_i$. Furthermore, since $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$, we have $\bar{p}_i \in \mathcal{S}_{\bar{\Phi}^\kappa_i}$.

In conclusion, we have $\forall i \in 1..n.\ (\bar{v}_i = \bullet \Rightarrow \bar{p}_i = \bar{p}'_i \wedge \bar{p}_i \in \mathcal{S}_{\bar{\Phi}^\kappa_i}) \wedge (\bar{v}_i \neq \bullet \Rightarrow \bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}^\kappa_i} \bar{p}'_i)$. Hence, it holds that $\bar{p} \xrightarrow{\bar{v},a}_{\mathcal{P}^\kappa} \bar{p}'$. □

Soundness of the analysis Proposition 3.4.16 formally describes the analysis technique. To show the soundness of Proposition 3.4.16, we have to prove that a branching bisimulation relation B between the κ-extended pattern networks of a transformation rule system implies, via state vector mappings, a branching bisimulation relation C between arbitrary original and transformed LTS networks.

As the κ-extended pattern networks represent abstractions from the networks they are mapped on, the relation B can be seen as an abstract relation between states of $\mathcal{N}$ and $T_\Sigma(\mathcal{N})$. For the matched local states, i.e., the matched states in the local process LTSs of the network, the relation is explicitly defined. In addition to this, the κ-state represents all unmatched local states.

A consequence of Lemma 3.4.15 is that two cases can be distinguished. If all process-local transitions are transformed, it follows that the state vector mapping preserves the branching structure of transitions enabled by non-κ-synchronisation laws. If no process-local transitions are transformed, it is still possible that a state $\bar{s}$ is related via C to a state $\bar{t}$ that is matched by at least one non-glue state; e.g., $\exists i \in 1..n, \bar{q}_i \in \mathcal{S}_{\mathcal{R}} \setminus \mathcal{S}_{\mathcal{L}}.\hat{m}_i(\bar{q}_i) = \bar{t}_i$. If $\bar{s}$ is able to perform an a-transition enabled by a $(\bar{v}, a) \in \mathcal{V}$, then $\bar{t}$ must be able to simulate this transition. Some local states $\bar{t}_i$ $(i \in Ac(\bar{v}))$ of $\bar{t}$ may be matched on by non-glue states. In this case, $\bar{t}$ is not able to perform the a-transition itself. Therefore, there must be a τ-path from $\bar{t}$ to a state $\hat{\bar{t}}$ such that $\hat{\bar{t}}$ can perform this a-transition as is shown in Lemma 3.4.14.

Recall that we assume that Σ has n rules, and that a rule R_i $(i \in 1..n)$ matches on Π_i in the LTS network that Σ is applied to. For a single transformation step, the rules in R can be reordered according to this assumption with an appropriate projection of the rule system. For confluent rule systems, the result can be lifted to confluent sequences of transformations steps and APC1' can be *weakened* again to APC1.

Proposition 3.4.16. *Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system satisfying ANC1, APC1', APC2, APC3, and APC4. Let $\bar{M}$ be a vector of match pairs of size n such that $m_i : \mathcal{L}_i \to \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \to T(\Pi_i)$ for all $i \in 1..n$.*

Then,

$$(\forall P \in \mathbb{D}.\ \bar{\mathcal{L}}^{\kappa,P} \underline{\leftrightarrow}_b \bar{\mathcal{R}}^{\kappa,P}) \implies \mathcal{N} \underline{\leftrightarrow}_b T_{\bar{M}}(\mathcal{N})$$

Proof. By definition, we have $\mathcal{N} \underline{\leftrightarrow}_b T_{\bar{M}}(\mathcal{N})$ iff there exists a branching bisimulation relation C with $\mathcal{I}_{\mathcal{N}}\ C\ \mathcal{I}_{T_{\bar{M}}(\mathcal{N})}$. Branching bisimilarity is a congruence for the construction of system LTSs from LTS networks. Therefore, since $\forall P \in \mathbb{D}.\ \bar{\mathcal{L}}^{\kappa,P} \underline{\leftrightarrow}_b \bar{\mathcal{R}}^{\kappa,P}$, by congruence, there is a relation B such that $\bar{\mathcal{L}}^{\kappa}\ B\ \bar{\mathcal{R}}^{\kappa}$. We define C as follows:

$$\begin{aligned} C = \{(\bar{s},\bar{t})\ |\exists \bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^{\kappa}}, \bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^{\kappa}}.\ \bar{p}\ B\ \bar{q} \wedge \bar{p} \vdash \bar{s} \wedge \bar{q} \vdash \bar{t} \\ \wedge \forall i \in 1..n.\ (\bar{p}_i = \kappa \vee \bar{q}_i = \kappa) \Rightarrow \bar{s}_i = \bar{t}_i\} \end{aligned}$$

To prove the proposition we have to show that C is a bisimulation relation. This requires proving that C relates the initial states of $\mathcal{N}$ and $T_{\bar{M}}(\mathcal{N})$ and that C satisfies Definition 2.2.2.

- *C relates the initial states of $\mathcal{N}$ and $T_{\bar{M}}(\mathcal{N})$.* We have $\mathcal{I}_{\mathcal{N}} = \mathcal{I}_{T_{\bar{M}}(\mathcal{N})}$. Hence, it suffices to show $\forall \bar{s} \in \mathcal{I}_{\mathcal{N}}.\ \bar{s}\ C\ \bar{s}$. Take an arbitrary state $\bar{s} \in \mathcal{I}_{\mathcal{N}}$, then by Lemma 3.4.11, there is a state $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^{\kappa}}$ with $\bar{p} \vdash \bar{s}$. Since $\bar{s} \in \mathcal{I}_{\mathcal{N}}$, it follows from Definition 3.3.3 that $\forall i \in 1..n.\ \bar{p}_i \in \mathcal{E}_{\mathcal{L}_i} \vee \bar{p}_i = \kappa$, i.e., $\bar{p} \in \mathcal{I}_{\bar{\mathcal{L}}^{\kappa}}$. By Lemma 3.4.9, we have $\bar{p}\ B\ \bar{p}$. It follows that $\bar{s}\ C\ \bar{s}$.

- *If $\bar{s}\ C\ \bar{t}$ and $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ then either $a = \tau \wedge \bar{s}'\ C\ \bar{t}$, or $\bar{t} \xrightarrow{\tau}{}^{*}_{T_{\bar{M}}(\mathcal{N})} \hat{\bar{t}} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}' \wedge \bar{s}\ C\ \hat{\bar{t}} \wedge \bar{s}'\ C\ \bar{t}'$.* Consider synchronisation law $(\bar{v}, a) \in \mathcal{V}$ enabling the transition $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$. Since $\bar{s}\ C\ \bar{t}$, there exist $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^{\kappa}}$ and $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^{\kappa}}$ such that $\bar{p}\ B\ \bar{q}$, $\bar{p} \vdash \bar{s}$, and $\bar{q} \vdash \bar{t}$, and $\forall i \in 1..n.\ (\bar{p}_i = \kappa \vee \bar{q}_i = \kappa) \Rightarrow \bar{s}_i = \bar{t}_i$ (3.2) . Furthermore, by Lemma 3.4.11, there is a state $\bar{p}' \in \mathcal{S}_{\bar{\mathcal{L}}^{\kappa}}$ with $\bar{p}' \vdash \bar{s}'$. We distinguish two cases:

  1. $\exists i \in Ac(\bar{v}).\ m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^{\kappa}} \bar{p}'_i$. We shall first establish that $(\bar{v}, a) \in \mathcal{V}'$, after which we can apply Lemma 3.4.15 to obtain a the corresponding a-transition in $\mathcal{T}_{\bar{\mathcal{L}}^{\kappa}}$. Assume for a contradiction that $(\bar{v}, a) \notin \mathcal{V}'$. Since there is an $i \in Ac(\bar{v})$ with $\bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^{\kappa}} \bar{p}'_i$, the $\bar{v}_i$ action must be a member of the actions of $\mathcal{L}_i$, i.e., $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$. However, since $i \in Ac(\bar{v})$ and $(\bar{v}, a) \in \mathcal{V} \setminus \mathcal{V}'$, by APC4, it must hold that $\bar{v}_i \notin \mathcal{A}_{\mathcal{L}_i}$ Hence, by contradiction, we have $(\bar{v}, a) \in \mathcal{V}'$. Now, by Lemma 3.4.15, there is a transition $\bar{p} \xrightarrow{a}_{\bar{\mathcal{L}}^{\kappa}} \bar{p}'$ enabled by $(\bar{v}, a)$. Since $\bar{p}\ B\ \bar{q}$, by Definition 2.2.2, we have the following two cases:
     - $a = \tau$ and $\bar{p}'\ B\ \bar{q}$. To show $\bar{s}\ C\ \bar{t}$, all ingredients but one are there. In particular, we still need to show that $\forall i \in 1..n.\ \bar{p}'_i = \kappa \vee \bar{q}_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}_i = \kappa$. Since $\mathcal{V}^{\kappa}$ and $\mathcal{V}'$ are disjoint, we must have $(\bar{v}, a) \notin \mathcal{V}^{\kappa}$. If $\bar{q}_i = \kappa$, then by Lemma 3.4.8, $\bar{p}_i = \kappa$. Since $\bar{p}_i = \kappa$ or $\bar{p}'_i = \kappa$ and $(\bar{v}, a) \notin \mathcal{V}^{\kappa}$, it follows that $i \notin Ac(\bar{v})$. Hence, by Definition 2.3.3, $\bar{s}'_i = \bar{s}_i$. Finally by (3.2), $\bar{s}_i = \bar{t}_i$.
     - $\bar{q} \xrightarrow{\tau}{}^{*}_{\bar{\mathcal{R}}^{\kappa}} \hat{\bar{q}} \xrightarrow{a}_{\bar{\mathcal{R}}^{\kappa}} \bar{q}'$ with $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$. From Lemma 3.4.13, it follows that there is a state $\hat{\bar{t}} \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\hat{\bar{q}} \vdash \hat{\bar{t}}$, a τ-path $\bar{t} \xrightarrow{\tau}{}^{*}_{T_{\bar{M}}(\mathcal{N})} \hat{\bar{t}}$, and $\forall i \in 1..n.\ \hat{\bar{q}}_i = \kappa \Rightarrow \hat{\bar{t}}_i = \bar{t}_i$ (3.3) . Furthermore, by Lemma 3.4.12, we have a state $\bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\bar{q}' \vdash \bar{t}'$, a transition $\hat{\bar{t}} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, and $\forall i \in 1..n.\ \bar{q}'_i = \kappa \Rightarrow \bar{t}'_i = \hat{\bar{t}}_i$ (3.4) . What remains to be shown is that $\bar{s}\ C\ \hat{\bar{t}}$ and $\bar{s}'\ C\ \bar{t}'$.

- For $\bar{s}\ C\ \hat{\bar{t}}$, all that is left to show is $\forall i \in 1..n.\ \bar{p}_i = \kappa \vee \hat{\bar{q}}_i = \kappa \Rightarrow \bar{s}_i = \hat{\bar{t}}_i$. Consider an $i \in 1..n$ with $\bar{p}_i = \kappa \vee \hat{\bar{q}}_i = \kappa$. By Lemma 3.4.7, $\hat{\bar{q}}_i = \kappa$ iff $\bar{q}_i = \kappa$. Hence, $\bar{p}_i = \kappa \vee \bar{q}_i = \kappa$. From (3.2), it follows that $\bar{s}_i = \bar{t}_i$. Finally, by Lemma 3.4.8, $\bar{p}_i = \kappa$ iff $\hat{\bar{q}}_i = \kappa$ and it follows from (3.3) that $\hat{\bar{t}}_i = \bar{t}_i$. Therefore, $\bar{s}_i = \bar{t}_i = \hat{\bar{t}}_i$. In conclusion, $\bar{s}\ C\ \hat{\bar{t}}$.
- Similarly, for $\bar{s}'\ C\ \bar{t}'$, all that is left to show is $\forall i \in 1..n.\ \bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}'_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$. By Lemma 3.4.6, $\hat{\bar{q}}_i = \kappa$ iff $\bar{q}'_i = \kappa$. Hence, $\bar{p}'_i = \kappa \vee \hat{\bar{q}}_i = \kappa$. From $\bar{s}\ C\ \hat{\bar{t}}$, it follows that $\bar{s}_i = \hat{\bar{t}}_i$. Finally, by Lemma 3.4.8, $\bar{p}'_i = \kappa$ iff $\bar{q}'_i = \kappa$ and it follows from (3.4) that $\bar{t}'_i = \hat{\bar{t}}_i$. Therefore, $\bar{s}'_i = \hat{\bar{t}}_i = \bar{t}'_i$. In conclusion, $\bar{s}'\ C\ \bar{t}'$.

2. $\neg 1$. Because $\neg 1$, we have $\neg\exists i \in Ac(\bar{v}).\ m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^\kappa} \bar{p}'_i$. That is, for all $i \in Ac(\bar{v})$ there is no transition in $\mathcal{T}_{\mathcal{L}_i}$ matching on $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$, or more formally, $\neg(m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^\kappa} \bar{p}'_i)$. Therefore, for all states $\bar{p}, \bar{p}' \in \mathcal{S}_{\bar{\mathcal{L}}^\kappa}$ with $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$, there is no transition $\bar{p} \xrightarrow{a}_{\bar{\mathcal{L}}^\kappa} \bar{p}'$. Moreover, by Definitions 3.3.5 and 3.3.3, for each $i \in 1..n$ state $\bar{p}_i$ is either a κ-state or an exit-state (in $\mathcal{E}_{\mathcal{L}_i}$), and state $\bar{p}'_i$ is either a κ-state or an in-state (in $\mathcal{I}_{\mathcal{L}_i}$), i.e., $\forall i \in Ac(\bar{v}).\ (\bar{p}_i \in \mathcal{I}_{\mathcal{L}_i} \vee \bar{p}_i \in \mathcal{E}_{\mathcal{L}_i}) \wedge (\bar{p}'_i \in \mathcal{I}_{\mathcal{L}_i} \vee \bar{p}'_i = \kappa)$ (3.5) . By applying Lemma 3.4.14, we get states $\hat{\bar{q}}, \bar{q}' \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$ such that there is a τ-path $\bar{q} \xrightarrow{\tau}{}^*_{\bar{\mathcal{R}}^\kappa} \hat{\bar{q}}$ with related states $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$, and the states have the following two properties: $\forall i \in Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i$ (3.6) , and $\forall i \in 1..n \setminus Ac(\bar{v}).\ \hat{\bar{q}}_i = \bar{q}'_i$ (3.7) .

   From Lemma 3.4.13 it follows that there is a state $\hat{\bar{t}} \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\hat{\bar{q}} \vdash \hat{\bar{t}}$, a τ-path $\bar{t} \xrightarrow{\tau}{}^*_{T_{\bar{M}}(\mathcal{N})} \hat{\bar{t}}$, and $\forall i \in 1..n.\ \hat{\bar{q}}_i = \kappa \Rightarrow \hat{\bar{t}}_i = \bar{t}_i$ (3.8) . We construct a state $\bar{t}' := \hat{\bar{t}}[\bar{s}'_i \mid i \in Ac(\bar{v})]$. By construction of $\bar{t}'$ we have $\forall i \in Ac(\bar{v}).\ \bar{t}'_i = \bar{s}'_i$ and $\forall i \in 1..n \setminus Ac(\bar{v}).\ \bar{t}'_i = \hat{\bar{t}}_i$. To prove that $\hat{\bar{t}} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, what remains to be shown is $\forall i \in Ac(\bar{v}).\ \hat{\bar{t}}_i = \bar{s}_i$: consider an $i \in Ac(\bar{v})$. By (3.6), $\hat{\bar{q}}_i = \bar{p}_i$. If $\hat{\bar{q}}_i = \kappa$, then also $\bar{p}_i = \kappa$ (Lemma 3.4.8). Therefore, by (3.8) and (3.2), $\hat{\bar{t}}_i = \bar{t}_i = \bar{s}_i$. If $\hat{\bar{q}}_i \neq \kappa$, then also $\bar{p}_i \neq \kappa$ (Lemma 3.4.8). It follows that $m_i(\bar{p}_i) = \bar{s}_i$ and $\hat{m}_i(\hat{\bar{q}}_i) = \hat{\bar{t}}_i$. By (3.5) and $\bar{p}_i \neq \kappa$, it holds that $\bar{p}_i \in \mathcal{E}_{\mathcal{L}_i}$. Thus, $\hat{m}_i(\bar{p}_i) = \bar{s}_i$ (by Definition 3.3.4). Finally, by injectivity of $\hat{m}$ we have $\hat{\bar{t}}_i = \bar{s}_i$.

   Hence, $\bar{t} \xrightarrow{\tau}{}^*_{T_{\bar{M}}(\mathcal{N})} \hat{\bar{t}} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$. What is left to show is $\bar{q}' \vdash \bar{t}'$, $\bar{s}\ C\ \hat{\bar{t}}$ and $\bar{s}'\ C\ \bar{t}'$.

   - For $\bar{q}' \vdash \bar{t}'$, we have to show that $\forall i \in 1..n.\ \bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$ (i.e., $\bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$) and $\forall i \in 1..n.\ (\bar{q}'_i \neq \kappa \implies \hat{m}_i(\bar{q}'_i) = \bar{t}'_i) \wedge (\bar{q}'_i = \kappa \implies \neg\exists x \in \mathcal{S}_{\mathcal{R}_i}.\ \hat{m}_i(x) = \bar{t}'_i)$. Consider an $i \in 1..n$. Based on the construction of $\bar{t}'$ we distinguish the following cases:
     - $i \in Ac(\bar{v})$. We have $\bar{t}'_i = \bar{s}'_i$ (by construction of $\bar{t}'$) and $\bar{q}'_i = \bar{p}'_i$ (by (3.6)). By (3.5) and $\bar{q}'_i = \bar{p}'_i$, either $\bar{q}'_i \in \mathcal{I}_{\mathcal{L}_i}$ or $\bar{q}'_i = \kappa$.
       If $\bar{q}'_i \in \mathcal{I}_{\mathcal{L}_i}$, then also $\bar{p}'_i \in \mathcal{I}_{\mathcal{L}_i}$ and we have $m_i(\bar{p}'_i) = \bar{s}'_i$ which we may rewrite to $\hat{m}_i(\bar{p}'_i) = \bar{s}'_i$ (by Definition 3.3.4). Finally, by $\bar{q}'_i = \bar{p}'_i$ and $\bar{t}'_i = \bar{s}'_i$, we have $\hat{m}_i(\bar{q}'_i) = \bar{s}'_i = \bar{t}'_i$. Furthermore, since $\hat{m}_i(\bar{q}'_i) = \bar{t}'_i$, it follows that $\bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$.
       If $\bar{q}'_i = \kappa$, then we have to show $\neg\exists x \in \mathcal{S}_{\mathcal{R}_i}.\ \hat{m}_i(x) = \bar{t}'_i$. Assume for a contradiction that there is a state $x \in \mathcal{S}_{\mathcal{R}_i}$ such that $\hat{m}_i(x) = \bar{t}'_i$. Since $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$ is not matched on, by Definition 3.3.3, we must have

$x \in \mathcal{I}_{\mathcal{L}_i}$. By $x \in \mathcal{I}_{\mathcal{L}_i}$ and $\bar{t}'_i = \bar{s}'_i$ (by construction of $\bar{t}'$), we have $m_i(x) = \hat{\bar{t}}_i$. However, since $\bar{q}'_i = \bar{p}'_i$ (by (3.6)), we have $\bar{p}'_i = \kappa$. Thus, by Definition 3.4.10, there is no such x with $m_i(x) = \hat{\bar{t}}_i$. Furthermore, since $\bar{s}'$ is not matched on by m_i, the state remains present in $T(\Pi_i)$. Hence, since $\bar{t}'_i = \bar{s}'_i$, it holds that $\bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$.

  - $i \notin Ac(\bar{v})$. We have $\bar{t}'_i = \hat{\bar{t}}_i$ (by construction of $\bar{t}'$) and $\hat{\bar{q}}_i = \bar{q}'_i$ (by (3.7)). The proof now follows directly by substituting $\bar{t}'_i$ for $\hat{\bar{t}}_i$ and substituting $\hat{\bar{q}}_i$ for $\bar{q}'_i$ in $\hat{\bar{q}} \vdash \hat{\bar{t}}$.

- For $\bar{s}\ C\ \hat{\bar{t}}$, we still have to show that $\forall i \in 1..n.\ \bar{p}_i = \kappa \vee \hat{\bar{q}}_i = \kappa \Rightarrow \bar{s}_i = \hat{\bar{t}}_i$. Consider an $i \in 1..n$ with $\bar{p}_i = \kappa \vee \hat{\bar{q}}_i = \kappa$. By Lemma 3.4.8, $\hat{\bar{q}}_i = \kappa$ iff $\bar{p}_i = \kappa$. Hence, by (3.8), we have $\hat{\bar{t}}_i = \bar{t}_i$. Finally, by (3.2) $\bar{s}_i = \bar{t}_i$, thus we have $\hat{\bar{t}}_i = \bar{t}_i = \bar{s}_i$. In conclusion, $\bar{s}\ C\ \hat{\bar{t}}$.
- Similarly, for $\bar{s}'\ C\ \bar{t}'$, all that is left to show is that $\forall i \in 1..n.\ \bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}'_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$. If $i \in Ac(\bar{v})$, then by construction of $\bar{t}'$, we have $\bar{s}'_i = \bar{t}'_i$. Conversely, if $i \notin Ac(\bar{v})$, then $\bar{t}'_i = \hat{\bar{t}}_i$ and $\bar{s}'_i = \bar{s}_i$. Hence, also $\bar{q}'_i = \hat{\bar{q}}_i$ and $\bar{p}'_i = \bar{p}_i$. Since $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$, it now follows that $\bar{p}_i = \kappa \vee \hat{\bar{q}}_i = \kappa$. By $\bar{s}\ C\ \hat{\bar{t}}$, we have $\bar{s}_i = \hat{\bar{t}}_i$, thus $\bar{s}'_i = \bar{s}_i = \hat{\bar{t}}_i = \bar{t}'_i$. In conclusion, $\bar{s}'\ C\ \bar{t}'$.

- *The symmetric case: if* $\bar{s}\ C\ \bar{t}$ *and* $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ *then either* $a = \tau \wedge \bar{s}\ C\ \bar{t}'$, *or* $\bar{s} \xrightarrow{\tau}{}^*_{\mathcal{N}} \hat{\bar{s}} \xrightarrow{a}_{\mathcal{N}} \bar{s}' \wedge \hat{\bar{s}}\ C\ \bar{t} \wedge \bar{s}'\ C\ \bar{t}'$. This case is symmetric to the previous case, with the exception that $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \hat{\bar{t}}$ is enabled by some $(\bar{v}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. Therefore, when transition $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is not matched on, we have to show that $(\bar{v}, a) \in \mathcal{V}$. Let $\bar{t}, \bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ such that $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is enabled by some $(\bar{v}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. Furthermore, transition $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is not matched on. Assume for a contradiction that $(\bar{v}, a) \in \hat{\mathcal{V}}$. Since $(\bar{v}, a) \in \hat{\mathcal{V}}$ is introduced by the transformation, by APC2, for all $i \in Ac(\bar{v})$ the action $\bar{v}_i$ does not occur in the original process Π_i, i.e., for all $i \in 1..n$, we have $\bar{v}_i \notin \mathcal{A}_i$. Hence, these actions $\bar{v}_i$ must be introduced by $\mathcal{R}_i$, i.e., $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i} \setminus \mathcal{A}_{\mathcal{L}_i}$. It follows that there is a transition matching $\bar{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, contradicting our earlier assumption. Hence, we must have $(\bar{v}, a) \in \mathcal{V}$. □

Completeness of the analysis In the next proposition, it is expressed that our analysis technique is complete. This means that the analysis will always report that the left and right κ-extended pattern networks of a rule system Σ are branching bisimilar if for any input network $\mathcal{N}$ on which Σ is applicable and any given matching it holds that $\mathcal{N} \leftrightarrows_b T_{\bar{M}}(\mathcal{N})$.

Similarly to the analysis of a single transformation rule this analysis considers all input LTS networks that satisfy the analysis and application conditions. Hence, even when a rule system does *not* preserve a given property it may still be the case that the property is preserved for some instances of the transformation. For instance, given a rule system Σ that is not property preserving there may be an input network $\mathcal{N}'$ with a vector of matches $\bar{M}'$ such that $\mathcal{N}' \leftrightarrows_b T_\Sigma(\mathcal{N}')$. However, it is guaranteed for Σ that there also exists an LTS network $\mathcal{N}''$ and vector of matches $\bar{M}''$ such that $\mathcal{N}'' \not\leftrightarrows_b T_\Sigma(\mathcal{N}'')$.

Proposition 3.4.17. *Consider a rule system* $\Sigma = (R, \mathcal{V})$. *Let* $\mathbb{M}$ *be the set of all LTS networks and* $\Sigma_{\mathcal{N}}$ *be the set of all possible vectors* $\bar{M}$ *of* n *match pairs defining a transformation step using* Σ *for an LTS network* $\mathcal{N} = (\Pi, \mathcal{V}) \in \mathbb{M}$ *of size* n. *Such a vector*

$\bar{M}$ consists of tuples $(m_i, \hat{m}_i)$, with $m_i : \mathcal{L}_i \to \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \to T(\Pi_i)$, respectively. The following holds:

$$(\forall \mathcal{N} \in \mathbb{M}, \bar{M} \in \Sigma_{\mathcal{N}}.\ \mathcal{N} \underline{\leftrightarrow}_b T_{\bar{M}}(\mathcal{N})) \implies \bar{\mathcal{L}}^{\kappa} \underline{\leftrightarrow}_b \bar{\mathcal{R}}^{\kappa}$$

Proof. Assume that for all $\mathcal{N} \in \mathbb{M}$ and $\bar{M} \in \Sigma_{\mathcal{N}}$ it holds that $\mathcal{N} \underline{\leftrightarrow}_b T_{\bar{M}}(\mathcal{N})$. Trivially, we have $\bar{\mathcal{L}}^{\kappa} \in \mathbb{M}$ and trivial matches $(m_i : \bar{\mathcal{L}}_i \to \bar{\mathcal{L}}_i^{\kappa}, \hat{m}_i : \bar{\mathcal{R}}_i \to T(\bar{\mathcal{L}}_i^{\kappa}))$ (for each $i \in 1..n$) constituting a vector of matches $\bar{M}$. It follows from the assumption that $\bar{\mathcal{L}}^{\kappa} \underline{\leftrightarrow}_b T_{\Sigma}(\bar{\mathcal{L}}^{\kappa})$. By Definition 3.4.4, $\bar{\mathcal{L}}^{\kappa} = (\langle \mathcal{L}_1^{\kappa}, \ldots, \mathcal{L}_n^{\kappa} \rangle, \mathcal{V}' \cup \mathcal{V}'^{\kappa})$ and $\bar{\mathcal{R}}^{\kappa} = (\langle \mathcal{R}_1^{\kappa}, \ldots, \mathcal{R}_n^{\kappa} \rangle, \mathcal{V}' \cup \hat{\mathcal{V}} \cup \mathcal{V}'^{\kappa} \cup \hat{\mathcal{V}}^{\kappa})$. By Definition 3.4.3, we have $T_{\Sigma}(\bar{\mathcal{L}}^{\kappa}) = (\langle \mathcal{R}_1^{\kappa} \ldots \mathcal{R}_n^{\kappa} \rangle, \mathcal{V}' \cup \mathcal{V}'^{\kappa} \cup \hat{\mathcal{V}} \cup \hat{\mathcal{V}}^{\kappa}) = \bar{\mathcal{R}}^{\kappa}$. It follows that $\bar{\mathcal{L}}^{\kappa} \underline{\leftrightarrow}_b \bar{\mathcal{R}}^{\kappa}$. □

3.5 Experiments

The verification technique presented in this chapter is implemented in a tool called Refiner [218]. Refiner is implemented in Python 3 and can be run from the command-line. It is platform-independent, and allows performing transformations of LTS networks, and checking semantics and property preservation. It integrates with the action-based, explicit-state model checking toolsets Cadp [82] and mCRL2 [54]. These tools can be used to specify and verify concurrent systems.

Given an LTS network $\mathcal{N}$ in Cadp's Exp format and a rule system Σ specified in Refiner's Rs format, Refiner first checks whether Σ is applicable to the LTS network as discussed in Section 3.4.2.2. If Σ is applicable to $\mathcal{N}$, then Refiner transforms the given LTS network into a new LTS network $T_{\Sigma}(\mathcal{N})$ (also in Exp format) by exhaustively applying transformation steps as presented in Definition 3.4.3. If the rule system is not applicable to the LTS network, then Refiner reports an error indicating which of the application conditions (APC1', APC2, APC3, or APC4) is violated. Depending on the condition the problematic match, transformation rule, process LTS, action label, and/or transition is reported.

The verification of a rule system Σ given in Refiner's rs format is performed as described in Section 3.4.2.1. First, Refiner checks whether Σ is confluent (ANC1). Then, the κ-extended rule system is constructed. Finally, Refiner uses the mCRL2 tool LtsCompare to perform bisimilarity comparisons with an implementation of the GJKW algorithm [91].

For the experiments in this section both the old version (v1) of Refiner, using the theory of previous works [59, 214, 217, 218], and the new version (v2) of Refiner, using the theory presented in this chapter, were used. The new theory shows that the number of checks Refiner performs can be reduced from $2^n - 1$ checks per set of dependent transformation rules to one check per set of dependent rules.

For the experiments presented in this section Refiner was compiled using Nuitka to C++ to reduce performance overhead caused by the Python virtual machine.[6] We ran Refiner on the standard machines of the DAS-5 cluster [17], which have an Intel Haswell E5-2630-v3 2.4 GHz processor, 64 GB memory, running CentOS Linux 7.2. Each experiment was conducted no longer than 80 hours and aborted in case the machine ran out of memory.

[6] `http://nuitka.net`

We have performed two types of experiments. The *first* setup compares traditional model checking with the transformation verification approach presented in this work. The results are reported in Section 3.5.1. The *second* setup aims to compare the previous transformation verification algorithm (Refiner v1) with the algorithm presented in this chapter (Refiner v2). Those results are discussed in Section 3.5.2. [7]

3.5.1 Comparing Traditional Model Checking and Transformation Verification

The *goal* of this experimental setup is to compare the running time of transformation verification with traditional model checking. For model checking we have selected the explicit-state model checker Cadp. For the transformation verification we use Refiner with the algorithms presented in this chapter.

We have selected a set of base models for verification and transformation. Each base model, say $\mathcal{N}$ 1, was transformed using Refiner resulting in a new model $\mathcal{N}$ 2. For two cases we have applied two different rule systems to the base model, the models are then called $\mathcal{N}$ 2A and $\mathcal{N}$ 2B. Another two cases were transformed even further resulting in a model $\mathcal{N}$ 3.

Each of the models is verified for the absence of deadlocks. Likewise, the rule systems are verified for the preservation of absence of deadlocks, i.e., the rule systems may not introduce new deadlocks.

Each base model was verified using Cadp and the verification time was measured. The rule systems were applied and verified by Refiner and both the transformation and verification time were measured. After each transformation the resulting model was verified again using Cadp.

For both tools we have measured the wall clock time (i.e., the real elapsed time) using the Unix time command:

```
/usr/bin/time -f "%e" <tool>
```

The argument `-f "%e"` specifies that the time written as output should follow format `"%e"` where `%e` indicates the wall clock time. The time is measured for `<tool>`, the command used to invoke the given tool.

Invocation of the tools For traditional model checking, the Cadp 2016-k tool Evaluator was used. All models (before and after transformation) were verified using the Cadp toolkit. The rule system application and verification was performed using Refiner. To enable replication of the experiment, we record and explain the commands performed to conduct the experiments.

The command used to verify a network using Cadp is:

```
exp.open <network>.exp evaluator <property>.mcl
```

The `exp.open` tool reads the input model and the `evaluator` tool subsequently checks on-the-fly whether the given μ-calculus formula described in `<property>.mcl` is satisfied.

[7] All models used in the experiments are available at `http://www.win.tue.nl/mdse/property_preservation/FAC2017_experiments.zip`.

We have verified a rule system <rule_system> with respect to a given property <property> with Refiner using the following command:

```
refiner -q -c2 -c <rule_system> -p <property> -f
```

The argument -q indicates that Refiner should run in quiet mode, i.e., no messages are sent to the standard output. The -c2 argument tells Refiner to use the verification algorithm presented in this chapter. Next, -c <rule_system> indicates that Refiner will verify the rule system <rule_system>. Finally, -p <property> specifies the property that Refiner verifies to be preserved, and -f indicates that Kooman's fair abstraction rule [122] holds for the considered system. Without the -f argument, Refiner checks for divergence preserving branching bisimilarity [207, 217] of the rule networks to determine whether liveness properties are preserved. Conversely, if -f is enabled, branching bisimulation checking is used instead for safety properties and inevitable reachability properties.

A network <network> was transformed using a rule system <rule_system> with Refiner as follows:

```
refiner -n <network> -r <rule_system>
```

The argument -n <network> specifies the network <network> used as input for the transformation. To apply the rule system <rule_system> to the network the argument -r <rule_system> is used.

The set of test cases As test input, we selected nine case studies, two newly created ones, three from the set of mCRL2 models distributed with its toolset, and four from the set of Cadp models.

The newly created ones are the following:

1. ABP is a model consisting of six independent subsystems, each involving two processes communicating using the Alternating Bit Protocol.
2. Broadcast consists of ten independent subsystems, each containing three processes that together synchronise in a three-party synchronisation.

The models stemming from the mCRL2 toolset distribution are the following:

1. The 1394-fin model describes the 1394 or firewire protocol. It has been created by Luttik [141].
2. The ACS model describes a part of the software of the Alma project of the European Southern Observatory, which involves controlling a large collection of radio telescopes. It consists of a manager and some containers and components. The model was created by Ploeger [169].
3. Wafer stepper is a model of a wafer stepper.

Finally, the Cadp models are the following:

1. ODP is a model of an open distributed processing trader [84].
2. The DES model describes an implementation of the data encryption standard, which allows to cipher and decipher 64-bit vectors using a 64-bit key vector [153].

Table 3.1: Experimental results: verification of various models using on-the-fly verification in CADP and transformation verification in REFINER with running times in seconds

Name	On-the-fly verif. (CADP)			Trans. & verif. (REFINER)		
	#States	Running time	φ holds	Trans. time	Verif. time	Check
ACS 1	3,484	0.98	✓	n.a.	n.a.	✓
ACS 2	21,936	4.95	✓	0.50	0.18	✓
1394-fin 1	198,692	6.93	✓	n.a.	n.a.	✓
1394-fin 2	6,679,222	152.43	✓	3.63	0.18	✓
Wafer stepper 1	78,919	7.82	✓	n.a.	n.a.	✓
Wafer stepper 2	474,457	51.38	✓	0.15	0.18	✓
ODP 1	91,394	13.85	✓	n.a.	n.a.	✓
ODP 2	7,699,456	62.16	✓	0.31	0.18	✓
DES 1	64,498,297	739.54	✓	n.a.	n.a.	✓
DES 2	64,498,317	795.21	✓	1137.21	0.17	✓
Broadcast 1	1,024	43.67	✓	n.a.	n.a.	✓
Broadcast 2A	30,654,053	982.53	✓	0.01	0.17	✗
Broadcast 2B	60,466,176	2,130.53	✓	0.06	0.17	✓
ABP 1	759,375	15.90	✓	n.a.	n.a.	✓
ABP 2A	380,204,032	13,256.61	✗	0.09	0.18	✗
ABP 2B	656,356,768	28,182.56	✓	0.10	0.18	✓
HAVi-LE 1	15,688,570	292.50	✓	n.a.	n.a.	✓
HAVi-LE 2	190,208,728	3,675.75	✓	0.71	0.58	✓
HAVi-LE 3	3,048,589,069	167,070.35	✓	0.67	0.18	✓
Erat. Sieve 1	6,539,813	2,003.78	✓	n.a.	n.a.	✓
Erat. Sieve 2	19,434,968	6,056.11	✓	23.76	0.17	✓
Erat. Sieve 3	135,159,971	42,449.26	✓	23.97	0.17	✓

3. HAVi-LE describes the asynchronous Leader Election protocol used in the HAVi (Home Audio-Video) standard, involving three device control managers. The model is fully described by Romijn [177].

4. Erat. Sieve is a specification of a distributed Eratothenes sieve. It consists of a number generator and a chain of 17 units, each unit i filtering out the i^{th} prime number.

Each model was subjected to one or two transformations, of the following types: (1) adding internal computations, (2) adding support for lossy channels by introducing the Alternating Bit Protocol (the ABP case), and (3) breaking down broadcast, i.e., synchronisations involving more than two parties to combinations of two-party synchronisations (the broadcast and the HAVi leader election case).

Discussion of results Table 3.1 presents the experimental results. The *first* column indicates the name of a test model. For each model, the number at the end of each name reflects the order in which the models were obtained, i.e., original models are indexed by '1'. Models resulting from the application of a transformation to the corresponding original model are indexed by '2', '2A, or '2B'. The '2A'- and '2B'-models are independently obtained via two different transformations from the corresponding '1'-model. Models indexed by '3' are likewise the result of transforming the corresponding '2'-model.

In the *On-the-fly verif. (*CADP*)* column metrics obtained from CADP's on-the-fly verification on the test model are displayed. We report the number of states each state

space consists of (*#States* column), and the running time (in seconds) to generate and verify these using CADP (*Running time* column).

The *Trans. & verif. (*REFINER*)* column shows the running time (in seconds) of applying (*Trans. time* column) and verifying (*Verif. time* column) the rule system using REFINER. The running time of the transformation is the required time to obtain a particular model by applying a rule system. Because the base models are not the results of the application of a rule system there is no transformation and verification time. Therefore, the time measurements are not applicable, indicated with "n.a.", for base models. Note that REFINER does not actually check the state spaces of the models indexed by '2' and '3', but instead can reason about their correctness by verifying the applied transformation rules.

Finally, the φ *holds* and *Check* columns provides the outcome of the verification for each case for CADP and REFINER, respectively. For CADP ✓ indicates that the LTS network satisfies the property and ✗ indicates that it does not. For REFINER ✓ indicates that application of the rule system preserves the property and ✗ indicates that it might not.

The transformations applied on the Broadcast case attempt to translate a three-party synchronisation to a protocol where one of the three processes performs a handshaking protocol with the other two processes before continuing. The transformation of Broadcast 2A omits a time-out transition returning to a previous state which causes the branching bisimulation comparison to identify a non-inert τ-transition. The transformed model still satisfies the property, despite that REFINER concludes that it may not be preserved in general.

The first transformation from to ABP 2A introduces a deadlock. Due to a typo a transition that should have send an 'F' sends a 'T' instead causing a deadlock in the protocol. Thus, ABP 2A does not satisfy deadlock-absence and REFINER also concludes that the corresponding transformation does not preserved the property in general.

In terms of running times obtaining DES 2 is quite costly. The network of DES 1 contains one particularly large LTS, consisting of more than four million states, making transformation at least as costly as verifying DES 1. In fact, it is even slower, but this is due to the fact that CADP reads compressed LTS files, while REFINER does not, hence the latter requires more time to read the input network.

The experiments demonstrate that preservation checking with REFINER is many orders of magnitude faster compared to verifying the property again, if the state space is of reasonable size. This is not surprising, as the check only focuses on the applied change, not the resulting state space. In our benchmark set of examples, the changes can be verified practically instantaneously, resulting in most verification tasks being ready in 0.17 or 0.18 seconds on our test machines. If one would compare REFINER's running times with those of other model checkers, the conclusion would be the same.

The usual workflow of verifying a model and verifying and applying the corresponding transformations is as follows. First, the initial model (version 1 in Table 3.1) is verified, using a model checker such as CADP. Then, instead of applying a transformation and then verifying the resulting model again, one can verify the transformation itself. If the transformation does not preserve the desired property, then the model resulting from the application of the transformation (version 2A in Table 3.1) must be verified. In case verification of the transformation produces a positive result, it can be safely applied without having to verify the resulting model (versions 2, 2B and 3 in Table 3.1).

3.5.2 Refiner v1 Versus Refiner v2

The *goal* of this experiment is to compare the running times of the previous version of the algorithm (Refiner v1) and the algorithm presented in this chapter (Refiner v2). For this we have generated a scalable set of rule systems that model the transformation of a token-ring.

We measured the time both Refiner versions spent building and verifying the state space. The state space construction and verification algorithms are the only difference between Refiner v1 and v2. Therefore, the elements that are equivalent for both versions are eliminated from the measurements. Although the state space generation and verification dominate the running time, for the small models, incorporating tasks such as reading the input may introduce unnecessary noise. By removing this noise we are able to observe the direct impact the new algorithm has on the performance of the tool.

For time measurement we used the Python method `time.time()`. This is sufficiently accurate, even for the smaller models, as it can measure differences of even less than a hundredth of a second between the Refiner versions.

We ran both Refiner v1 and v2 in quiet mode to limit the time spent writing messages to the standard output. Refiner v1 needs to check all κ-extended pattern networks of subsets of the set of transformation rules. Refiner v1 can distribute the checks over several cores to increase the performance. For completeness sake, for Refiner v1 the experiments were run with both a single thread and multiple threads (eight in the case of a standard DAS-5 machine). The former allows a better comparison of the theoretical performance improvements as Refiner v2 only uses a single thread. The latter allows a more practical comparison showing the typical performance of Refiner v1 in its common use.

The largest check performed by Refiner v1 considers the entire set of transformation rules when the left and right κ-extended pattern networks are checked for branching bisimilarity. This largest check is equivalent to the check proposed in this work and performed by Refiner v2. This is the result of improved theoretical results, as presented in the current chapter, that proved that only this largest check is required. Hence, the expectation is that Refiner v1 will never perform better than Refiner v2.

Invocation of tools All generated rule systems were verified for full semantic preservation using single-threaded Refiner v1, multi-threaded Refiner v1, and Refiner v2. For reproducibility of the experiment, we explain the commands used for this experiment below.

For Refiner v1 using a single thread the following command was used:

```
refiner -q -t 1 -c <rule_system>
```

The argument `-q` indicates that Refiner should run in quiet mode, i.e., no messages are sent to the standard output. The maximum number of threads is set using the `-t` argument. Here, `-t 1` expresses that only a single thread is used. Argument `-c <rule_system>` tells Refiner to verify the rule system `<rule_system>`. In this experimental setup, the models are named `gen_i` with $i \in 2..n$.

For Refiner v1 using multiple threads we used the command:

```
refiner -q -c <rule_system>
```

Without the `-t` argument Refiner creates a thread for each core of the machine and distributes the checks over these threads. In the case of a standard DAS-5 machine

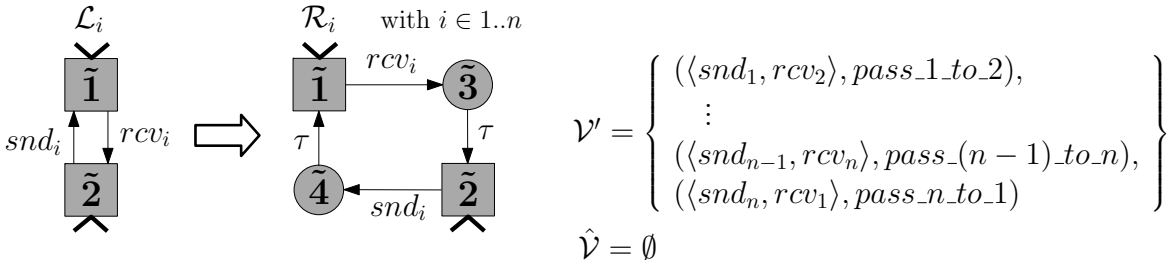


Figure 3.13: Rule system transforming a token ring of size n

eight threads are used. The remaining arguments are the same as the ones for the single threaded variant.

REFINER v2 is invoked using the following command:

```
refiner -q -c2 -c <rule_system>
```

The `-c2` argument sets a flag telling REFINER to use the REFINER v2 algorithm. The remainder of the arguments is the same as for the REFINER v1 experiments.

Generation of rule systems We have generated rule systems consisting of a specified number of rules n. The smallest rule system generated contains two transformation rules. The number of rules is incremented by one until a rule system is generated for which the verification exceeds the maximum time of 80 hours or the machine runs out of memory (64 GB).

The rule systems considered for this experiment model the transformation of token rings of size n. The network topology of a token ring ensures that the rule system consists of one dependency set. A generic representation of these rule systems is shown in Figure 3.13. For a generated rule system of size n there are n transformation rules and n synchronisation laws. The action-labels snd_i and rcv_i indicate that the i^{th} rule or node performs a *send* and *receive* action, respectively.

The transformation rules introduce an extra τ-transition directly after the snd_i and rcv_i transitions. These τ-transitions represent internal computation; for instance, when the token is received a node may need to process the data before sending it to the next node.

The synchronisation laws describe the passing of the token from the current node (snd_i) to the next node (rcv_{i+1}), represented by a $pass\_i\_to\_(i+1)$-action (where $i \in 1..(n-1)$). The last synchronisation law specifies that the last node passes the token (snd_n) back to the first node (rcv_1). Hence, the rule system describes the transformation of a token ring consisting of n nodes.

Discussion of results The results of this experiment are presented in Table 3.2. The size n of the rule system model is indicated by the *first* column. Each row shows the results of the verification for the rule system model of size n. The *State space of* $\bar{\mathcal{R}}^{\kappa}$ column describe the size of the right κ-extended pattern network in terms of number of states (*#States* column) and transitions (*#Transitions* column). The right κ-extended pattern network is the larger of the two networks, therefore, it gives a good indication of the size of the state space. The *Trans. verif. running times (*REFINER*)* column presents the averaged running time per model in seconds for single-threaded REFINER v1,

Table 3.2: Experimental results: verification of token ring rule systems of size n using Refiner v1 and Refiner v2; running times are shown in seconds

n	State space of $\bar{\mathcal{R}}^\kappa$		Trans. verif. running times (Refiner)		
	#States	#Transitions	v1 1 thread	v1 8 threads	v2
2	24	67	0.06	0.04	0.03
3	124	486	0.14	0.06	0.06
4	624	3,173	0.35	0.16	0.15
5	3,124	19,608	1.46	0.61	0.54
6	15,624	116,967	6.97	2.69	2.47
7	78,124	680,298	36.77	18.49	14.45
8	390,624	3,881,545	227.83	148.81	85.35
9	1,953,124	21,816,540	1,467.08	1,111.45	517.14
10	9,765,624	121,162,769	10,287.18	8,138.29	3,522.03

multi-threaded Refiner v1, and Refiner v2 in the *v1 1 thread*, *v1 8 threads*, and *v2* columns, respectively. The running time, shown in seconds, is the average running of ten runs.

For the rule system with $n = 11$ the machines ran out of memory (64 GB). The memory consumption is dominated by the state space of the κ-extended pattern networks. The number of states of the system LTS of this model is 48,828,124.

The results show that for all Refiner versions the running time increases exponentially, as does the state space of the considered checks. This is due to the exponential blow up of the state spaces of $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$. Of these two state spaces $\bar{\mathcal{R}}^\kappa$ is significantly larger because of the two τ-transitions.

The results clearly show that the algorithm presented in this chapter (Refiner v2) outperforms both the single- and multi-threaded variant of the previous version of the algorithm (Refiner v1). As mentioned before, this is no surprise as the largest check performed by Refiner v1 is the same as the check performed by Refiner v2. The extra checks that Refiner v1 performs consider the projected rule systems of all subsets of dependent transformation rules. These projected rule systems become exponentially smaller as the size of the subset decreases, thereby also decreasing the size of the state space analysed in the check that is performed. The decreasing size of these extra checks explains why the running time of Refiner v1 is only a few factors larger than the running time of Refiner v2.

A last observation we can make based on Table 3.2 is that the running time ratio of Refiner v1 to v2 seems to increase. To investigate whether this is a trend we have plotted the ratios between the different Refiner versions in Figure 3.14. The *horizontal axis* depicts the number of rules in the generated rule system, the *vertical axis* indicates the ratio. Although the number of data points is limited, the graph gives us some insights into the practical running time improvements.

The ratio of Refiner v1 with a single thread to Refiner v1 with eight threads is shown as the *continuous line* where the diamonds indicate the data points. The ratio shows a general decline towards 1 as n grows, i.e., for large n the benefit of the extra threads becomes negligible. This is unexpected as more cores should be able to verify more checks in the dependency set simultaneously. Upon further inspection we found that the utilisation of the cores was not efficient. Refiner performs smaller checks before larger checks. Hence, the largest check is performed last. Thus, in the worst case, the remaining cores are not utilised when this final check is performed.

For the same reason there is a sudden decline in the ratio from a token ring transfor-

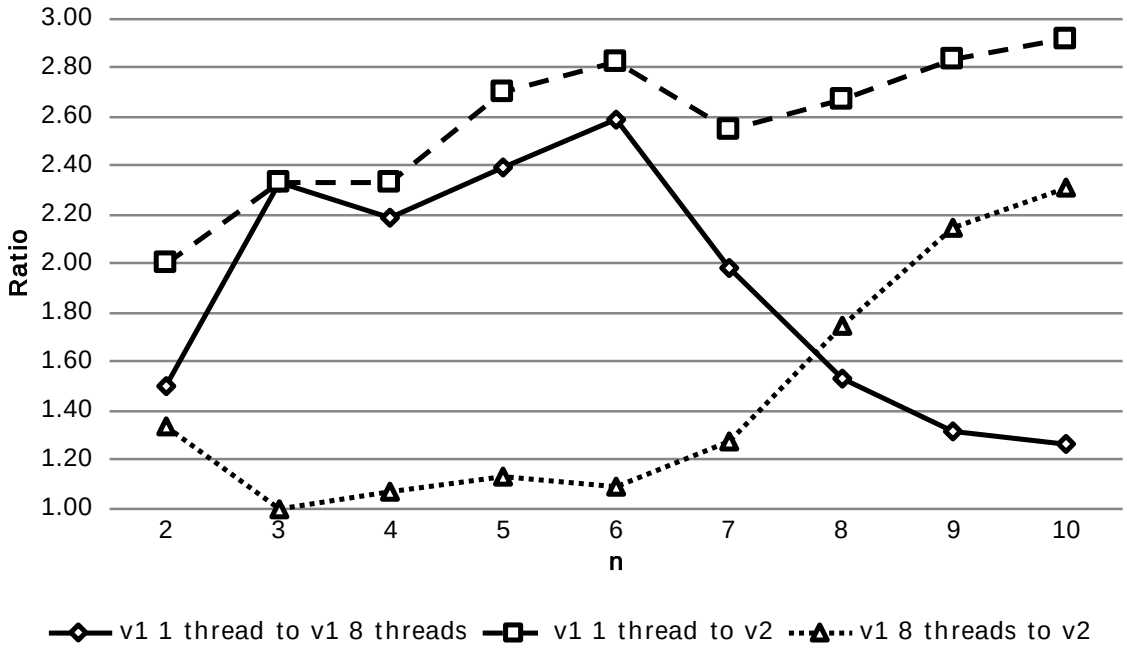


Figure 3.14: Ratios between the Refiner versions when analysing the transformation of a token ring of size n

mation with three rules to one with four rules. At three rules, there are exactly eight checks, thus the eight cores are utilised optimally. Whereas at four rules, sixteen checks need to be performed, but cores are poorly utilised as the larger checks are performed last. Finally, at two rules, there are only four checks while there are eight cores available. As not all the cores can be put to use only a small performance gain is obtained . We choose not to optimise the distribution of checks over the available cores for Refiner v1 as Refiner v2 is by definition more efficient.

The *dashed line* shows the ratio of the single threaded Refiner v1 to Refiner v2 where the data points are indicated with squares. The ratio increases as n grows. Due to the limited number of data points we cannot estimate the trend function. The running time analysis predicts an exponential trend, however, but this is not visible in the data.

The *dotted line* shows the ratio of Refiner v1 running 8 threads to Refiner v2 where the data points are indicated with triangles. This ratio shows an increase as n grows similar to the ratio between the single threaded Refiner v1 and Refiner v2. As n grows the data points move towards the dashed line (the single threaded Refiner v1 to Refiner v2 ratio). This is expected as the difference between the single threaded Refiner v1 and multi-threaded Refiner v1 decreases as n grows as indicated by the continuous line. At three rules, there are exactly as many cores as there are checks for Refiner v1. Hence, at this point the performance of the multi-threaded Refiner v1 is equivalent to that of Refiner v2. However, at two rules, there are more cores than checks, but Refiner v2 performs better than Refiner v1. As the checks are extremely small for rule systems consisting of 2 rules it is likely that the overhead of the threads have a visible impact on the performance of Refiner v1.

3.6 Conclusions

We discussed the correctness of an LTS transformation verification technique. The aim of the technique is to verify whether a given LTS transformation system Σ preserves a property φ, written in a fragment of the μ-calculus, for all possible input models formalised as LTS networks. It does this by determining whether Σ is guaranteed to transform an input network into one that is branching bisimilar, ignoring the behaviour not relevant for φ.

We demonstrated the efficiency of the verification technique compared to model checking the entire model again after it has been transformed. Many orders of magnitude speed-up can be achieved through model transformation verification.

We improved upon previous results by reducing the number of required bisimulation checks from $2^n - 1$ per set of dependent transformation rules to one per set of dependent rules. Experimentally, we demonstrated that our new verification algorithm outperforms the previous one, even if the latter uses eight threads and the new one only a single thread.

Furthermore, the expressiveness of transformation rules was extended by distinguishing between glue-states that allow incoming and/or outgoing transitions that enter or leave the LTS pattern, respectively. This work presents a proof for these results. The proof has been verified in Coq.

The property preservation check is limited to rule systems that adhere to the applicability and admissibility conditions. Input networks must be admissible as well. Furthermore, application of a rule system to an input network must satisfy application conditions APC1 to APC4.

Even when a transformation does not preserve a given property, it may still be possible that said property holds for the output model of a specific instance of the transformation. Nevertheless, only transformations that are property-preserving can be reused without the need for additional verification.

Future work In earlier work, we used branching bisimulation *with explicit divergence* [207, 217], which preserves τ-loops and therefore liveness properties. In future work, we would like to prove that for this flavour of bisimulation the technique is also correct. Moreover, we would like to investigate what the practical limitations of the pre-conditions of the technique are in industrial sized transformation systems.

Although the current implementation of the property-preservation check approach does allows the retrieval of counter-examples when a transformation is deemed not property-preserving, it would be beneficial te project these counter examples onto the designed rule system. Such counter-examples help the developers understand *why* their transformation may not preserve a given property. This information guides the developer in fixing the issue.

Our framework can be extended in a number of ways to reason about additional aspects of concurrent systems. For instance, in line with the encoding proposed in [213], timing information could be included in the LTSs to design timed systems and express transformations of timed behaviour. This would also introduce the possibility to analyse the impact a transformation will have on the performance of a system under transformation [219], by means of timed branching bisimulation checking [76]. The capability to reason about system performance could be further strengthened by also introducing probabilities on the LTS transitions [16]. Existing tools, such as PRISM [129] and extensions [32], could then be employed to conduct the analysis of the systems. An interesting

challenge is then how to involve these probabilities in the verification of transformations as well.

Wijs [214] proposed an extension to the transformation verification technique that explicitly considers the communication interfaces between components, thereby removing the completeness condition ANC1 regarding synchronising behaviour being transformed (see Section 3.4.1). The correctness of such an extension follows from the fact that branching bisimulation and divergence-preserving branching bisimulation are congruences for LTS networks (see Chapter 4). For this a consistent decomposition, as presented in Chapter 4, must be implemented

Finally, when refining of an LTS network it may be necessary to refine the property as well. This could be achieved by verifying whether a transformed network satisfies some property ψ if the network the transformation is applied to satisfies another property φ. Such a verification method verifies pre- and post-properties on transformations. This verification method may have other applications as well; e.g., modelling of loop invariants as transformations. However, before all of its applications can be investigated, we will have to investigate the working and limitations of the method first. In the next section we briefly conjecture about the verification of pre- and post-properties on transformations.

Chapter 4

Compositional Model Checking is Lively

Compositional model checking approaches attempt to limit state space explosion by iteratively combining behaviour of some of the components in the system and reducing the result modulo an appropriate equivalence relation. For an equivalence relation to be applicable, it should be a congruence for parallel composition where synchronisations between the components may be introduced.

An equivalence relation preserving both safety and liveness properties is divergence-preserving branching bisimulation (DPBB). It is generally assumed that DPBB is a congruence for parallel composition, even in the context of synchronisations between components. However, so far, no such results have been published. This work finally proves that this is the case.

We also show that DPBB is a congruence for LTS networks in which many LTSs are composed in parallel at once with support for multi-party synchronisation. Additionally, we discuss how to safely decompose an existing LTS network in components such that the re-composition is equivalent to the original LTS network.

Finally, to demonstrate the effectiveness of compositional model checking with intermediate DPBB reductions, we discuss the results we obtained after having conducted a number of experiments.

This chapter is taken from

[58] de Putter, S., Lang, F., and Wijs, A. Compositional Model Checking is Lively - Extended Version. *Science of Computer Programming* (2019). Special Issue on FACS. *Manuscript under review*

a special issue extension of

[64] de Putter, S., and Wijs, A. J. Compositional Model Checking Is Lively. In *FACS* (2017), vol. 10487 of *LNCS*, Springer, pp. 117–136

which received the FACS 2017 *Best Student Paper Award*

4.1 Introduction

Model checking [16, 45] is one of the most successful approaches for the analysis and verification of the behaviour of concurrent systems. However, a major issue is the so-called *state space explosion problem*: the state space of a concurrent system tends to increase exponentially as the number of parallel processes increases linearly. Often, it is difficult or infeasible to verify realistic large scale concurrent systems. Over time, several methods have been proposed to tackle the state space explosion problem. Prominent approaches are the application of some form of on-the-fly reduction, such as Partial Order Reduction [165] or Symmetry Reduction [43], and compositional verification, for instance using Compositional Reasoning [47] or Partial Model Checking [8, 9, 135].

The key operations in compositional approaches are the composition and decomposition of systems. First, a system is decomposed into two or more components. Then, one or more of these components are manipulated (e.g., reduced). Finally, the components are re-composed. Comparison modulo an appropriate equivalence relation is applied to ensure that the manipulations preserve properties of interest (for instance, expressed in the modal μ-calculus [123]). These manipulations are sound if and only if the equivalence relation is a congruence for the composition expression.

Two prominent equivalence relations are branching bisimulation and divergence-preserving branching bisimulation (DPBB) [204, 207].[1] Branching bisimulation preserves safety properties, while DPBB preserves both safety and liveness properties.

Van Glabbeek, Luttik, and Trčka [206] show that DPBB is the coarsest equivalence contained in divergence sensitive branching bisimulation equivalence that is a congruence for parallel composition. However, compositional reasoning requires equivalences that are a congruence for parallel composition where new *synchronisations between parallel components* may be introduced, which is not considered by Van Glabbeek, Luttik, and Trčka. It is known that branching bisimulation is a congruence for parallel composition of synchronising Labelled Transition Systems (LTSs), this follows from the fact that parallel composition of synchronising LTSs can be expressed as a WB cool language [26]. However, obtaining such results for DPBB requires more work. To rigorously prove that DPBB is indeed a congruence for parallel composition of synchronising LTSs, a proof assistant, such as Coq [20], is required. So far, no results, obtained with or without the use of a proof assistant, have been reported.

A popular toolbox that offers a selection of compositional approaches is CADP [81]. CADP offers both *property-independent* approaches (e.g., compositional model generation, smart reduction, and compositional reasoning via behavioural interfaces) and *property-dependent* approaches (e.g., property-dependent reductions [144] and partial model checking [8]). The formal semantics of concurrent systems are described using *networks of LTSs* [133], or *LTS networks* for short. An LTS network consists of n LTSs representing the parallel processes. A set of synchronisation laws is used to describe the possible communication, i.e., synchronisation, between the process LTSs.

In this setting, this work considers parallel composition of synchronising LTS networks. Given two LTS networks $\mathcal{N}$ and $\mathcal{N}'$ of size n related via a DPBB relation B, another LTS network $\mathcal{P}$ of size m, and a parallel composition operator $\|_\sigma$ with a relation σ that specifies synchronization between components, we show there is a DPBB relation C such

[1] It should be noted that a distinction can be made between divergence-sensitive branching bisimulation [155] and branching bisimulation with explicit divergence, also known as divergence-preserving branching bisimulation [204, 207]. Contrary to the former, the latter distinguishes deadlocks and livelocks, and the latter is the coarsest congruence contained in the former.

that

$$\mathcal{N}\ B\ \mathcal{N}' \implies (\mathcal{N} \parallel_\sigma \mathcal{P})\ C\ (\mathcal{N}' \parallel_\sigma \mathcal{P}) \wedge (\mathcal{P} \parallel_\sigma \mathcal{N})\ C\ (\mathcal{P} \parallel_\sigma \mathcal{N}')$$

This result subsumes the composition of individual synchronising LTSs via composition of LTS networks of size one. Moreover, generalization to composition of multiple LTS networks can be obtained via a reordering of the processes within LTS networks.

Contributions In this work, we prove that DPBB is a congruence for parallel composition of LTS networks. From this it follows that DPBB is a congruence for parallel composition of synchronising LTSs. Furthermore, we present a method to safely decompose an LTS network in components such that the composition of the components is equivalent to the original LTS network. The proofs (with exception Proposition 4.6.1) have been mechanically verified using the Coq proof assistant and are available online. [2] The mechanical verification of the related proofs in this chapter gives us confidence in the correctness of Proposition 4.6.1.

Associativity and commutativity are desirable properties as they indicate that composition of LTS networks may be done in any order. Furthermore, from these properties it follows that DPBB is also a congruence for LTS networks as defined by Garavel, Lang, and Mateescu [81]. To this end we define a composition operator $\parallel$ for LTS networks that is both associative and commutative. The operator $\parallel$ is built from $\parallel_\sigma$ with a relation σ implementing synchronisation on the common alphabet.

Due to the definition of LTS networks $\parallel$ is not strictly commutative, however, we show that $\parallel$ is commutative with respect to global behaviour. In short, given LTS networks $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{O}$, operator $\parallel$ is associative, i.e.,

$$\mathcal{N} \parallel (\mathcal{P} \parallel \mathcal{O}) = (\mathcal{N} \parallel \mathcal{P}) \parallel \mathcal{O}$$

and commutative with respect to global behaviour, i.e.,

$$\mathcal{G}_\mathcal{N} \parallel \mathcal{G}_\mathcal{P} = \mathcal{G}_\mathcal{P} \parallel \mathcal{G}_\mathcal{N}$$

Moreover, we discuss an adaptation of the definition of LTS networks using indexed families for which $\parallel$ is truly commutative.

From associativity and commutativity of $\parallel$ it follows that DPBB is a congruence for LTS networks of which the set of synchronisation laws implements synchronisation on the common alphabet. However, it is unnecessary to require that the set of synchronisation laws implements synchronisation on the common alphabet. Furthermore, this requirement excludes many LTS networks in practice. Therefore, we present a proof that does not require synchronisation on a common alphabet. Given two networks $\mathcal{N}$ and $\mathcal{P}$ with vectors of LTSs Π and Ψ, respectively, of size n such that for each $i \in 1..n$ the i^{th} processes of the vectors are related by a DPBB relation B_i, we show that there is a DPBB relation C that relates the networks $\mathcal{N}$ and $\mathcal{P}$:

$$(\forall i \in 1..n.\ \Pi_i\ B_i\ \Psi_i) \implies \mathcal{G}_\mathcal{N}\ C\ \mathcal{G}_\mathcal{P}$$

Finally, we discuss the effectiveness of compositionally constructing state spaces with intermediate DPBB reductions in comparison with the classical, non-compositional state space construction. The discussion is based on results we obtained after having conducted

[2] `http://www.win.tue.nl/mdse/composition/DPBB_is_a_congruence_for_synchronizing_LTSs.zip`

a number of experiments using the CADP toolbox. Crouzen and Lang [56] report on experiments comparing the run-time and memory performance of three compositional verification techniques. As opposed to these experiments, our experiments concern the comparison of compositional and classical, non-compositional state space construction.

Structure of the chapter Related work is discussed in Section 4.2. Next, the formal composition of LTS networks is presented in Section 4.3. We prove that DPBB is a congruence for the composition of LTS networks. Section 4.4 is on the decomposition of an LTS network. Decomposition allows the redefinition of a system as a set of components. Section 4.5 introduces an instance of the composition operator that is both associative and commutative. From this operator it follows that DPBB is a congruence for LTS networks if the set of synchronisation laws implement synchronisation on the common alphabet. This restriction is lifted in Section 4.6. In Section 4.7 we apply the theoretical results to a set of use cases comparing a compositional construction approach with non-compositional state space construction. In Section 4.8 we present the conclusions and directions for future work.

4.2 Related Work

Networks of LTSs are introduced by Lang [132]. Lang mentions that strong and branching bisimulations are congruences for the operations supported by LTS networks. Among these operations is the parallel composition with synchronisation on equivalent labels. A proof for branching bisimulation has been verified in PVS by Jaco van de Pol and a textual proof was written, but both the textual proof and the PVS proof have not been made public [134]. An axiomatisation for a rooted version of divergence-preserving branching bisimulation has been performed in a Master graduation project [192]. However, the considered language does not include parallel composition. In this chapter, we formally show that DPBB is also a congruence for parallel composition with synchronisations between components. As DPBB is a branching bisimulation relation with an extra case for explicit divergence, the proof we present also formally shows that branching bisimulation is a congruence for parallel composition with synchronisations between components.

Another approach supporting compositional verification is presented by Lang [133]. Given an LTS network and a component selected from the network the approach automatically generates an interface LTS from the remainder of the network. This remainder of the network is called the environment. The interface LTS represents the synchronisation possibilities that are offered by the environment. This requires the construction and reduction of the system LTS of the environment. The advantage of this method is that transitions and states that do not contribute to the system LTS can be removed. In our approach only the system LTS of the considered component must be constructed. The environment is left out of scope until the components are composed.

Many process algebras support parallel composition with synchronisation on labels. Often a proof is given showing that some bisimulation is a congruence for these operators [54, 124, 143, 145]. However, to the best of our knowledge no such proofs exist considering DPBB. Furthermore, if LTSs can be derived from their semantics (such as is the case with Structural Operational Semantics) then the fact that DPBB is a congruence for such a parallel composition can be directly derived from our results.

To generalize the congruence proofs a series of meta-theories have been proposed for algebras with parallel composition [26, 201, 209]. Verhoef [209] introduces the *panth*

format. Verhoef shows that strong bisimulation is a congruence for algebras that adhere to the panth format. The focus of the work is on the expressiveness of the format. Bloom [26] proposes *WB cool* formats for four bisimulations: weak bisimulation, rooted weak bisimulation, branching bisimulation, and rooted branching bisimulation. It is shown that these bisimulations are congruences for the corresponding formats. Ulidowski and Phillips [201] propose similar formats for eager bisimulation and branching bisimulation. Eager bisimulation is a kind of weak bisimulation which is sensitive to divergence. The above mentioned formats do not consider DPBB. In our work we have shown that DPBB is a congruence for parallel composition of LTS networks and LTSs.

Wijs [214] proposes a decomposition for LTS transformation systems of LTS networks. The work aims to verify the transformation of a component that may synchronise with other components. The paper proposes to calculate so called detaching laws which are similar to our interface laws. The approach can be modelled with our method. In fact, we show that the derivation of these detaching laws does not amount to a desired decomposition, i.e., the re-composition of the decomposition is *not* equivalent to the original system (see Example 4.4.3 discussed in Section 4.4).

A projection of an LTS network given a set of indices is presented in [81]. Their projection operator is similar to the consistent decomposition of LTS networks that we proposed. In fact, with a suitable operator for the reordering of LTS networks our decomposition operation is equivalent to their projection operator. Furthermore, we show that admissibility properties of the LTS network are indeed preserved for such consistent decompositions.

4.3 Composition of LTS Networks

This section introduces the composition of these two LTS networks. The LTS network is defined in Definition 2.3.1. Composition of process LTSs results in a system LTS (Definition 2.3.3) that tends to grow exponentially when more processes are considered.

An LTS network can be seen as being composed of several *components*, each of which consists of a number of individual processes in parallel composition, with *intra-component* synchronisation laws describing how the processes inside a component should synchronise with each other. Furthermore, *inter-component* synchronisation laws define how the components as a whole should synchronise with each other. Compositional construction of a minimal version of the final system LTS may then be performed by first constructing the system LTSs of the different components, then minimising these, and finally combining their behaviour. Example 4.3.1 presents an example of a network with two components and an inter-component synchronisation law.

Example 4.3.1 (Component)**.** *Consider an LTS network* $\mathcal{N} = (\Pi, \mathcal{V})$ *with processes* $\Pi = \langle \Pi_1, \Pi_2, \Pi_3 \rangle$ *and synchronisation laws* $\mathcal{V} = \{(\langle a, \bullet, \bullet \rangle, a), (\langle \bullet, b, b \rangle, b), (\langle c, c, c \rangle, c)\}$. *We may split up the network in two* components*, say* $\mathcal{N}_1 = (\langle \Pi_1 \rangle, \mathcal{V}_1)$ *and* $\mathcal{N}_{\{2,3\}} = (\langle \Pi_2, \Pi_3 \rangle, \mathcal{V}_{\{2,3\}})$. *Then,* $(\langle c, c, c \rangle, c)$ *is an* inter-component law *describing synchronisation between* $\mathcal{N}_1$ *and* $\mathcal{N}_{\{2,3\}}$. *The component* $\mathcal{N}_1$ *consists of process* Π_1*, and the set of intra-component synchronisation laws* $\mathcal{V}_1 = \{(\langle a, \bullet, \bullet \rangle, a)\}$ *operating solely on* Π_1. *Similarly, component* $\mathcal{N}_{\{2,3\}}$ *consists of* Π_2 *and* Π_3*, and the set of intra-component synchronisation laws* $\mathcal{V}_{\{2,3\}} = \{(\langle \bullet, b, b \rangle, b)\}$ *operating solely on* Π_2 *and* Π_3.

The challenge of compositional construction is to allow manipulation of the components while guaranteeing that the observable behaviour of the system as a whole remains

equivalent modulo DPBB (Definition 2.2.3). Even though synchronisation laws of a component may be changed, we must somehow preserve synchronisations with the other components. Such a change of synchronisation laws occurs, for instance, when reordering the processes in a component, or renaming actions that are part of inter-component synchronisation laws.

In this chapter, we limit ourselves to composition of two components: a left and a right component. This simplifies notations and proofs. However, the approach can be generalised to splitting networks given two sets of indices indicating which processes are part of which component, i.e., a projection operator can be used to project distinct parts of a network into components.

In the remainder of this section, first, we formalise LTS networks composition. Then, we show that admissibility is preserved when two admissible networks are composed. Finally, we prove that DPBB is a congruence for composition of LTS networks.

Composing LTS networks Before defining the composition of two networks, we introduce a relation indicating how the inter-component laws should be constructed from the interfaces of the two networks. An inter-component law can then be constructed by combining the interface vectors of the components and adding a result action. This is achieved through a given *interface relation*, presented in Definition 4.3.2, relating interface actions to result actions.

Definition 4.3.2 (Interface Relation)**.** *Consider LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ of size n and m, respectively. An* interface relation *over $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$ is a relation $\sigma \subseteq \mathcal{A}_\mathcal{V} \setminus \{\tau\} \times \mathcal{A}_\mathcal{W} \setminus \{\tau\} \times \mathcal{A}$ describing how the interface actions of $\mathcal{N}_\Pi$ should be combined with interface actions of $\mathcal{N}_\Psi$, and what the action label should be resulting from successful synchronisation. The set $\mathcal{A}$ is the set of actions resulting from successful synchronisation between Π and Ψ. The actions mapped by σ are considered the interface actions.*

An interface relation implicitly defines how inter-component synchronisation laws should be represented in the separate components. These local representatives are called the *interface synchronisation laws*. An interface relation for $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ implies the following sets of interface synchronisation laws:

$$\mathcal{V}_\sigma = \{(\bar{v}, a) \in \mathcal{V} \mid (a, b, c) \in \sigma\}$$

$$\mathcal{W}_\sigma = \{(\bar{w}, b) \in \mathcal{W} \mid (a, b, c) \in \sigma\}$$

An interface synchronisation law makes a component's potential to synchronise with other components explicit. An interface synchronisation law has a synchronisation vector, called the *interface vector*, that may be part of inter-component laws. The result action of an interface synchronisation law is called an *interface action*. These notions are clarified further in Example 4.3.3.

Example 4.3.3 (Interface Vector and Interface Law)**.** *Let $\mathcal{N} = (\langle \Pi_1, \Pi_2, \Pi_3 \rangle, \mathcal{V})$ be a network with inter-component synchronisation law $(\langle a, a, b \rangle, c) \in \mathcal{V}$ and a component $M_{\{1,2\}} = (\langle \Pi_1, \Pi_2 \rangle, \mathcal{V}_{\{1,2\}})$. Then, $\langle a, a \rangle$ is an* interface vector *of $M_{\{1,2\}}$, and given a corresponding* interface action *α, the* interface law *is $(\langle a, a \rangle, \alpha)$.*

Together the interface laws and interface relation describe the possible synchronisations between two components, i.e., the interface laws and interface relation describe inter-component synchronisation laws. Given two sets of laws $\mathcal{V}$ and $\mathcal{W}$ and an interface

relation σ, the inter-component synchronisation laws are defined as follows:

$$\sigma(\mathcal{V}, \mathcal{W}) = \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v}, \alpha) \in \mathcal{V} \wedge (\bar{w}, \beta) \in \mathcal{W} \wedge (\alpha, \beta, a) \in \sigma\}$$

The interface relation suggests a partitioning of $\mathcal{V}$ and $\mathcal{W}$ into two sets of synchronisation laws: the interface and non-interface synchronisation laws. Given LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ of size n and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ of size m, the non-interface synchronisation laws are the sets $(\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet$ and ${}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$ of synchronisation laws $\mathcal{V} \setminus \mathcal{V}_\sigma$ padded with m $\bullet$'s and $\mathcal{W} \setminus \mathcal{W}_\sigma$ padded with n $\bullet$'s, respectively, where the padding of some set of synchronisation laws $\mathcal{X}$ is defined as follows

$$\mathcal{X}^\bullet = \{(\bar{v} \parallel \bullet^m, a) \mid (\bar{v}, a) \in \mathcal{X}\}$$

$${}^\bullet\mathcal{X} = \{(\bullet^n \parallel \bar{w}, a) \mid\mid (\bar{w}, a) \in \mathcal{X}\}$$

For instance, given a law $(\langle a, \bullet, b\rangle, c) \in \mathcal{X}$ the synchronisation law with 2 $\bullet$'s padded at the end is $(\langle a, \bullet, b, \bullet, \bullet\rangle, c) \in \mathcal{X}^\bullet$.

The application of the interface relation, i.e., formal composition of two LTS networks, is presented in Definition 4.3.4. We show that a component may be exchanged with a divergence-preserving branching bisimilar component iff the interface actions are not hidden. In other words, the interfacing with the remainder of the networks is respected when the interface actions remain observable.

Definition 4.3.4 (Composition of LTS networks)**.** *Consider LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ of size n and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ of size m. Let $\sigma \subseteq \mathcal{A}_\mathcal{V} \setminus \{\tau\} \times \mathcal{A}_\mathcal{W} \setminus \{\tau\} \times \mathcal{A}$ be an interface relation describing the synchronisations between $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$. The* composition *of $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$, denoted by $\mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi$, is defined as the LTS network $(\Pi \parallel \Psi, \mathcal{V} \parallel \mathcal{W})$, where $\mathcal{V} \parallel \mathcal{W} = (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet \cup {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma) \cup \sigma(\mathcal{V}, \mathcal{W})$.*

As presented in Proposition 4.3.5, LTS networks that are composed (according to Definition 4.3.4) from two admissible networks (Definition 2.3.2) are admissible as well. Therefore, composition of LTS networks is compatible with the compositional verification approaches of CADP [81].

Proposition 4.3.5. *Let $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ be admissible LTS networks of length n and m, respectively. Furthermore, let $\sigma \subseteq \mathcal{A}_\mathcal{V} \setminus \{\tau\} \times \mathcal{A}_\mathcal{W} \setminus \{\tau\} \times \mathcal{A}$ be an interface relation. Then, the network $\mathcal{N} = \mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi$, composed according to Definition 4.3.4, is also* admissible.

Proof. We show that $\mathcal{N}$ satisfies Definition 2.3.2:

- *No synchronisation and renaming of τ's.* Let $(\bar{v}, a) \in (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet \cup {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma) \cup \sigma(\mathcal{V}, \mathcal{W})$ be a synchronisation law with $\bar{v}_i = \tau$ for some $i \in 1..(n+m)$. We distinguish two cases:
  - $(\bar{v}, a) \in (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet \cup {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$. By construction of $(\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet$ and ${}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$, and admissibility of $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$, we have $\forall j \in 1..n.\ \bar{v}_j \neq \bullet \implies i = j$, $\forall j \in (n+1)..(n+m).\ \bar{v}_j \neq \bullet \implies i = j$ and $a = \tau$. Hence, it holds that $\forall j \in 1..(n+m).\ \bar{v}_j \neq \bullet \implies i = j$ (no synchronisation of τ's) and $a = \tau$ (no renaming of τ's).

– $(\bar{v}, a) \in \sigma(\mathcal{V}, \mathcal{W})$. By definition of $\sigma(\mathcal{V}, \mathcal{W})$, there are interface laws $(\bar{v}', \alpha') \in \mathcal{V}$ and $(\bar{v}'', \alpha'') \in \mathcal{W}$ such that $(\alpha', \alpha'', a) \in \sigma$. Hence, either $1 \leq i \leq n$ with $\bar{v}'_i = \tau$ or $n < i \leq n + m$ with $\bar{v}''_{i-n} = \tau$. Since $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$ are admissible, we must have $\alpha' = \tau$ or $\alpha'' = \tau$, respectively. However, the interface relation does not allow τ as interface actions, therefore, the proof follows by contradiction.

It follows that $\mathcal{N}$ does not allow synchronisation and renaming of τ's.

- *No cutting of τ's.* Let $(\Pi \parallel \Psi)_i$ be a process with $\tau \in \mathcal{A}_{(\Pi \parallel \Psi)_i}$ for some $i \in 1..(n+m)$. We distinguish the two cases $1 \leq i \leq n$ and $n < i \leq n + m$. It follows that $\tau \in \mathcal{A}_{\Pi_i}$ for the former case and $\tau \in \mathcal{A}_{\Psi_{i-n}}$ for the latter case. Since both $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$ are admissible and no actions are removed in $(\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet$ and ${}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$, in both cases there exists a $(\bar{v}, a) \in (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet \cup {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma) \cup \sigma(\mathcal{V}, \mathcal{W})$ such that $\bar{v}_i = \tau$. Hence, the composite network $\mathcal{N}$ does not allow cutting of τ's.

Since the three admissibility properties hold, the composed network $\mathcal{N}$ satisfies Definition 2.3.2. □

DPBB is a congruence for LTS network composition Proposition 4.3.6 shows that DPBB is a left-congruence for the composition of LTS networks according to Definition 4.3.4. The proof that DPBB is a right-congruence for the composition of LTS networks is symmetric. Hence, DPBB is a congruence for the composition of LTS networks according to Definition 4.3.4.

It is worth noting that an interface relation does not map τ's, i.e., synchronisation of τ-actions is not allowed. In particular, this means that interface actions must *not* be hidden when applying verification techniques to a component. Moreover, Proposition 4.3.6 subsumes the composition of single LTSs, via composition of LTS networks of size one with trivial sets of intra-component synchronisation laws.

Proposition 4.3.6. *Consider LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$, $\mathcal{N}_{\Pi'} = (\Pi', \mathcal{V}')$ of size n, and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ of size m. Let σ be an interface relation describing the coupling between the interface actions in $\mathcal{A}_\mathcal{V}$ and $\mathcal{A}_\mathcal{W}$. DPBB is a* congruence *for composition of LTS networks, i.e., it holds that*

$$\mathcal{N}_\Pi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'} \implies \mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'} \parallel_\sigma \mathcal{N}_\Psi$$

Proof. Intuitively, we have $\mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'} \parallel_\sigma \mathcal{N}_\Psi$ because $\mathcal{N}_\Pi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'}$ and the interface with $\mathcal{N}_\Psi$ is respected. Since $\mathcal{N}_\Pi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'}$, whenever a transition labelled with an interface action α in $\mathcal{N}_\Pi$ is able to perform a transition together with $\mathcal{N}_\Psi$, then $\mathcal{N}_{\Pi'}$ is able to simulate the interface α-transition and synchronise with $\mathcal{N}_\Psi$. It follows that the branching structure and divergence is preserved. For the sake of brevity we define the following shorthand notations: $\mathcal{N} = \mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi$ and $\mathcal{N}' = \mathcal{N}_{\Pi'} \parallel_\sigma \mathcal{N}_\Psi$. We show $\mathcal{N}_\Pi \leftrightarrow_b^\Delta \mathcal{N}_{\Pi'} \implies \mathcal{N} \leftrightarrow_b^\Delta \mathcal{N}'$.

Let B be a DPBB relation between $\mathcal{N}_\Pi$ and $\mathcal{N}_{\Pi'}$. By definition, we have $\mathcal{N} \leftrightarrow_b^\Delta \mathcal{N}'$ iff there exists a DPBB relation C such that $\mathcal{I}_\mathcal{N}\ C\ \mathcal{I}_{\mathcal{N}'}$. We define C as follows:

$$C = \{(\bar{s} \parallel \bar{r}, \bar{t} \parallel \bar{r}) \mid \bar{s}\ B\ \bar{t} \wedge \bar{r} \in \mathcal{S}_{\mathcal{N}_\Psi}\}$$

The component that is subject to change is related via the relation B that relates the states in Π and Π'. The unchanged component of the network is related via the shared state $\bar{r}$, i.e., it relates the states of Ψ to themselves.

To prove the proposition we have to show that C is a DPBB relation. This requires proving that C relates the initial states of $\mathcal{N}$ and $\mathcal{N}'$ and that C satisfies Definition 2.2.3.

- *C relates the initial states of $\mathcal{N}$ and $\mathcal{N}'$, i.e., $\mathcal{I}_{\mathcal{N}}\ C\ \mathcal{I}_{\mathcal{N}'}$.* We show that $\forall \bar{s} \in \mathcal{I}_{\mathcal{N}}.\ \exists \bar{t} \in \mathcal{I}_{\mathcal{N}'}.\ \bar{s}\ C\ \bar{t}$, the other case is symmetrical. Take an initial state $\bar{s} \parallel \bar{r} \in \mathcal{I}_{\mathcal{N}}$. Since $\mathcal{I}_{\mathcal{N}_\Pi}\ B\ \mathcal{I}_{\mathcal{N}_{\Pi'}}$ and $\bar{s} \in \mathcal{I}_{\mathcal{N}_\Pi}$, there exists a $\bar{t} \in \mathcal{I}_{\mathcal{N}_{\Pi'}}$ such that $\bar{s}\ B\ \bar{t}$. Therefore, we have $\bar{s} \parallel \bar{r}\ C\ \bar{t} \parallel \bar{r}$. Since $\bar{s} \parallel \bar{r}$ is an arbitrary state in $\mathcal{I}_{\mathcal{N}}$ the proof holds for all states in $\mathcal{I}_{\mathcal{N}}$. Furthermore, since the other case is symmetrical it follows that $\mathcal{I}_{\mathcal{N}}\ C\ \mathcal{I}_{\mathcal{N}'}$.

- *If $\bar{s}\ C\ \bar{t}$ and $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ then either $a = \tau \wedge \bar{s}'\ C\ \bar{t}$, or $\bar{t} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \hat{\bar{t}} \xrightarrow{a}_{\mathcal{N}'} \bar{t}' \wedge \bar{s}\ C\ \hat{\bar{t}} \wedge \bar{s}'\ C\ \bar{t}'$.* To better distinguish between the two parts of the networks, we unfold C and reformulate the proof obligation, with $\bar{s} = \bar{p} \parallel \bar{r}$ and $\bar{t} = \bar{q} \parallel \bar{r}$, as follows: *If $\bar{p}\ B\ \bar{q}$ and $\bar{p} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}} \bar{p}' \parallel \bar{r}'$ then either $a = \tau \wedge \bar{p}'\ B\ \bar{q} \wedge \bar{r} = \bar{r}'$, or $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \hat{\bar{q}} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}'} \bar{q}' \parallel \bar{r}' \wedge \bar{p}\ B\ \hat{\bar{q}} \wedge \bar{p}'\ B\ \bar{q}'$.* Consider synchronisation law $(\bar{v} \parallel \bar{w}, a) \in (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet \cup {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma) \cup \sigma(\mathcal{V}, \mathcal{W})$ enabling the transition $\bar{p} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}} \bar{p}' \parallel \bar{r}'$. We distinguish three cases:

  1. $(\bar{v} \parallel \bar{u}, a) \in (\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet$. It follows that $\bar{w} = \bullet^m$, and thus, subsystem $\mathcal{N}_\Psi$ does not participate. Hence, we have $\bar{r} = \bar{r}'$ and $(\bar{v}, a) \in \mathcal{V}$ enables a transition $\bar{p} \xrightarrow{a}_{\mathcal{N}_\Pi} \bar{p}'$. Since $\bar{p}\ B\ \bar{q}$, by Definition 2.2.3, we have:
     - $a = \tau$ with $\bar{p}'\ B\ \bar{q}$. Because $\bar{p}'\ B\ \bar{q}$ and $\bar{r} = \bar{r}'$, the proof trivially follows.
     - $\bar{q} \xrightarrow{\tau}{}^*_{\mathcal{N}_{\Pi'}} \hat{\bar{q}} \xrightarrow{a}_{\mathcal{N}_{\Pi'}} \bar{q}'$ with $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$. These transitions are enabled by laws in $\mathcal{V}' \setminus \mathcal{V}'_\sigma$. The set of derived laws are of the form $(\bar{v}' \parallel \bullet^m, \tau) \in (\mathcal{V}' \setminus \mathcal{V}'_\sigma)^\bullet$ enabling a τ-path from $\bar{q} \parallel \bar{r}$ to $\hat{\bar{q}} \parallel \bar{r}$, and there is a law $(\bar{v}' \parallel \bullet^m, a) \in (\mathcal{V}' \setminus \mathcal{V}'_\sigma)^\bullet$ enabling $\hat{\bar{q}} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}'} \bar{q}' \parallel \bar{r}$. Take $\bar{r}' := \bar{r}$ and the proof obligation is satisfied.
  2. $(\bar{v} \parallel \bar{w}, a) \in {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$. It follows that $\bar{v} = \bullet^n$, and thus, subsystems $\mathcal{N}_\Pi$ and $\mathcal{N}_{\Pi'}$ do not participate; we have $\bar{p} = \bar{p}'$ and $\bar{r} \xrightarrow{a}_{\mathcal{N}_\Psi} \bar{r}'$. We take $\bar{q}' := \bar{q}$. Hence, we can conclude $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \bar{q} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}} \bar{q}' \parallel \bar{r}'$, $\bar{p}\ B\ \bar{q}$, and $\bar{p}'\ B\ \bar{q}'$.
  3. $(\bar{v} \parallel \bar{w}, a) \in \sigma(\mathcal{V}, \mathcal{W})$. Both parts of the network participate in the transition $\bar{p} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}} \bar{p}' \parallel \bar{r}'$. By definition of $\sigma(\mathcal{V}, \mathcal{W})$, there are $(\bar{v}, \alpha) \in \mathcal{V}$, $(\bar{w}, \beta) \in \mathcal{W}$ and $(\alpha, \beta, a) \in \sigma$ such that $(\bar{v}, \alpha)$ enables a transition $\bar{p} \xrightarrow{\alpha}_{\mathcal{N}_\Pi} \bar{p}'$ and $(\bar{u}, \beta)$ enables a transition $\bar{r} \xrightarrow{\beta} \bar{r}'$. Since $\bar{p}\ B\ \bar{q}$, by Definition 2.2.3, we have:
     - $\alpha = \tau$ with $\bar{p}'\ B\ \bar{q}$. Since $\alpha \in \mathcal{A}_\mathcal{V} \setminus \{\tau\}$ we have a contradiction.
     - $\bar{q} \xrightarrow{\tau}{}^*_{\mathcal{N}'_{\Pi'}} \hat{\bar{q}} \xrightarrow{\alpha}_{\mathcal{N}'_{\Pi'}} \bar{q}'$ with $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}'\ B\ \bar{q}'$. Since τ actions are not mapped by the interface relation we have a set of synchronisation laws of the form $(\bar{v}' \parallel \bullet^m, \tau) \in (\mathcal{V}' \setminus \mathcal{V}'_\sigma)^\bullet$ enabling a τ-path $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \hat{\bar{q}} \parallel \bar{r}$.
       Let $(\bar{v}', \alpha) \in \mathcal{V}'$ be the synchronisation law enabling the α-transition. Since $(\alpha, \beta, a) \in \sigma$, α is an interface action and does not occur in $\mathcal{V}' \setminus \mathcal{V}'_\sigma$. It follows that $(\bar{v}', \alpha) \in \mathcal{V}'_\sigma$, and consequently $(\bar{v}' \parallel \bar{w}, a) \in \sigma(\mathcal{V}', \mathcal{W})$. Law $(\bar{v}' \parallel \bar{w}, a)$ enables the transition $\hat{\bar{q}} \parallel \bar{r} \xrightarrow{a}_{\mathcal{N}'} \bar{q}' \parallel \bar{r}'$, and the proof follows.

- *If $\bar{s}\ C\ \bar{t}$ and $\bar{t} \xrightarrow{a}_{\mathcal{N}'} \bar{t}'$ then either $a = \tau \wedge \bar{s}'\ C\ \bar{t}$, or $\bar{s} \xrightarrow{\tau}{}^*_{\mathcal{N}} \hat{\bar{s}} \xrightarrow{a}_{\mathcal{N}} \bar{s}' \wedge \bar{s}\ C\ \hat{\bar{t}} \wedge \bar{s}'\ C\ \bar{t}'$.* This case is symmetric to the previous case.

- *If* $\bar{s}\ C\ \bar{t}$ *and there is an infinite sequence of states* $(\bar{s}^k)_{k\in\omega}$ *such that* $\bar{s} = \bar{s}^0$, $\bar{s}^k \xrightarrow{\tau}_{\mathcal{N}} \bar{s}^{k+1}$ *and* $\bar{s}^k\ C\ \bar{t}$ *for all* $k \in \omega$, *then there exists a state* $\bar{t}'$ *such that* $\bar{t} \xrightarrow{\tau}^+_{\mathcal{N}'} \bar{t}'$ *and* $\bar{s}^k\ C\ \bar{t}'$ *for some* $k \in \omega$. Again we reformulate the proof obligation, with $\bar{s} = \bar{p} \parallel \bar{r}$ and $\bar{t} = \bar{q} \parallel \bar{r}$, to better distinguish between the two components : *if* $\bar{p} \parallel \bar{r}\ C\ \bar{q} \parallel \bar{r}$ *and there is an infinite sequence of states* $(\bar{p}^k \parallel \bar{r}^k)_{k\in\omega}$ *such that* $\bar{p} \parallel \bar{r} = \bar{p}^0 \parallel \bar{r}^0$, $\bar{p}^k \parallel \bar{r}^k \xrightarrow{\tau}_{\mathcal{N}} \bar{p}^{k+1} \parallel \bar{r}^{k+1}$ *and* $\bar{p}^k\ B\ \bar{q}$ *for all* $k \in \omega$, *then there exists states* $\bar{q}'$ *and* $\bar{r}'$ *such that* $\bar{q} \parallel \bar{r} \xrightarrow{\tau}^+_{\mathcal{N}'} \bar{q}' \parallel \bar{r}'$ *and* $\bar{p}^k\ B\ \bar{q}'$ *for some* $k \in \omega$.

  We distinguish two cases:

  1. All steps in the τ-sequence are enabled in $\mathcal{N}_\Pi$, i.e., $\forall k \in \omega.\ \bar{p}^k \xrightarrow{\tau}_{\mathcal{N}_\Pi} \bar{p}^{k+1}$. Since $\bar{p}\ B\ \bar{q}$, by condition 2.2.3 of Definition 2.2.3, it follows that there is a state $\bar{q}'$ with $\bar{q} \xrightarrow{\tau}{}^+ \bar{q}'$ and $\bar{p}^k\ B\ \bar{q}'$ for some $k \in \omega$. Since τ is not an interface action, the synchronization laws enabling $\bar{q} \xrightarrow{\tau}{}^+ \bar{q}'$ are also present in $\mathcal{N}'$. Hence, we have $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^+ \bar{q}' \parallel \bar{r}$ and $\bar{p}^k\ B\ \bar{q}'$ for $k \in \omega$.
  2. There is a $k \in \omega$ with $\neg \bar{p}^k \xrightarrow{\tau}_{\mathcal{N}_\Pi} \bar{p}^{k+1}$. We do have $\bar{p}^k \parallel \bar{r}^k \xrightarrow{\tau}_{\mathcal{N}} \bar{p}^{k+1} \parallel \bar{r}^{k+1}$ with $\bar{p}^k\ B\ \bar{q}$ (see antecedent at the start of the 'divergence' case). Since the τ-transition is not enabled in $\mathcal{N}_\Pi$ the transition must be enabled by a synchronisation law $(\bar{v} \parallel \bar{w}, \tau) \in {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma) \cup \sigma(\mathcal{V}, \mathcal{W})$. We distinguish two cases:
     - $(\bar{v} \parallel \bar{w}, \tau) \in {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$. The transition $\bar{p}^k \parallel \bar{r}^k \xrightarrow{\tau}_{\mathcal{N}} \bar{p}^{k+1} \parallel \bar{r}^{k+1}$ is enabled by $(\bar{v} \parallel \bar{w}, \tau) \in {}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$. Therefore, there is a transition $\bar{r}^k \xrightarrow{\tau}_{\mathcal{N}_\Psi} \bar{r}^{k+1}$ enabled by $(\bar{w}, \tau) \in \mathcal{W} \setminus \mathcal{W}_\sigma$. Since this transition is part of an infinite τ-sequence, there is a path $\bar{p} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \bar{p}^k \parallel \bar{r}^k$. Furthermore, condition 2.2.2 of Definition 2.2.3 holds for C, hence, there is a state $\bar{q}' \in \mathcal{S}_{\mathcal{N}_{\Pi'}}$ and a transition $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}_\Psi} \bar{q}' \parallel \bar{r}^k$ with $\bar{p}^k \parallel \bar{r}^k\ C\ \bar{q}' \parallel \bar{r}^k$. Therefore, we have $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^+_{\mathcal{N}'} \bar{q}' \parallel \bar{r}^{k+1}$. Finally, since $\bar{p}^k \parallel \bar{r}^k\ C\ \bar{q}' \parallel \bar{r}^k$, it follows that $\bar{p}^k\ B\ \bar{q}'$.
     - $(\bar{v} \parallel \bar{w}, \tau) \in \sigma(\mathcal{V}, \mathcal{W})$. By definition of $\sigma(\mathcal{V}, \mathcal{W})$, there are two laws $(\bar{v}, \alpha) \in \mathcal{V}$ and $(\bar{u}, \beta) \in \mathcal{W}$ with $(\alpha, \beta, \tau) \in \sigma$. The laws enable transitions $\bar{p}^k \xrightarrow{\alpha}_{\mathcal{N}_\Pi} \bar{p}^{k+1}$ and $\bar{r}^k \xrightarrow{\beta}_{\mathcal{N}_\Psi} \bar{r}^{k+1}$ respectively. Since $\bar{p}^k\ B\ \bar{q}$ and $\alpha \neq \tau$, by Definition 2.2.3, there are states $\hat{\bar{q}}, \bar{q}' \in \mathcal{S}_{\mathcal{N}_{\Pi'}}$ such that there is a sequence $\bar{q} \xrightarrow{\tau}{}^*_{\mathcal{N}_{\Pi'}} \hat{\bar{q}} \xrightarrow{\alpha}_{\mathcal{N}_{\Pi'}} \bar{q}'$ with $\bar{p}\ B\ \hat{\bar{q}}$ and $\bar{p}^{k+1}\ B\ \bar{q}'$. Let $(\bar{v}', \alpha) \in \mathcal{V}'$ be the law enabling the α-transition. Since $(\alpha, \beta, \tau) \in \sigma$, and consequently $(\bar{v}' \parallel \bar{w}, \tau) \in \sigma(\mathcal{X}', \mathcal{Y})$. Furthermore, the τ-path from $\bar{q}$ to $\hat{\bar{q}}$ is enabled by laws of the form $(\bar{v}'', \tau) \in \mathcal{V}' \setminus \mathcal{V}'_\sigma$. Hence, there is a series of transitions $\bar{q} \parallel \bar{r} \xrightarrow{\tau}{}^*_{\mathcal{N}'} \hat{\bar{q}} \parallel \bar{r}^k \xrightarrow{\tau}_{\mathcal{N}'} \bar{q}' \parallel \bar{r}^{k+1}$. Finally, recall that $\bar{p}^{k+1}\ B\ \bar{q}'$. Hence, also in this case the proof obligation is satisfied.

- *If* $\bar{p}\ C\ \bar{t}$ *and there is an infinite sequence of states* $(\bar{t}^k)_{k\in\omega}$ *such that* $\bar{t} = \bar{t}^0$, $\bar{t}^k \xrightarrow{\tau}_{\mathcal{N}'} \bar{t}^{k+1}$ *and* $\bar{p}\ C\ \bar{t}^k$ *for all* $k \in \omega$, *then there exists a state* $\bar{p}'$ *such that* $\bar{p} \xrightarrow{\tau}^+_{\mathcal{N}} \bar{p}'$ *and* $\bar{p}'\ C\ \bar{t}^k$ *for some* $k \in \omega$. This case is symmetric to the previous case. □

4.4 Decomposition of LTS Networks

In Section 4.3, we discuss the composition of LTS networks, in which a system is constructed by combining components. However, for compositional model checking

approaches, it should also be possible to correctly decompose LTS networks. In this case the inter-component laws are already known. Therefore, we can derive a set of interface laws and an interface relation specifying how the system is decomposed into components.

Consider the decomposition of an LTS network $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{Z})$ into components $\mathcal{P}$ and $\mathcal{O}$ according to some interface relation σ. First, the synchronisation laws $\mathcal{Z}$ are split into three disjoint sets: 1) $\mathcal{V}^\bullet$ the laws only applicable to the processes in Π; 2) ${}^\bullet\mathcal{W}$ the laws only applicable to the processes in Ψ; and 3) $\mathcal{X}$ the inter-component laws. Next, given two functions $f, g : \mathcal{X} \to \mathcal{A} \setminus \{\tau\}$ from inter-component laws to interface actions, the inter-component laws are decomposed into sets $\overleftarrow{\mathcal{X}} = \{(v, f(x)) \mid x \in \mathcal{X} \wedge x = (v \parallel w, a) \in \mathcal{X}\}$ and $\overrightarrow{\mathcal{X}} = \{(w, g(x)) \mid x \in \mathcal{X} \wedge x = (v \parallel w, a)\}$ of interface laws over Π and Ψ, respectively. Finally, the components are defined as $\mathcal{P} = (\Pi, \mathcal{V}^\bullet \cup \overleftarrow{X})$ and $\mathcal{O} = (\Psi, \mathcal{V}^\bullet \cup \overrightarrow{X})$.

To be able to apply Proposition 4.3.6 for compositional state space construction, the composition of the decomposed networks must be equivalent to the original system. If this holds we say a decomposition is *consistent* with respect to $\mathcal{N}$.

Definition 4.4.1 (Consistent Decomposition)**.** *Consider a network $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$ with $\mathcal{X}$ the set of inter-component laws. Say network $\mathcal{N}$ is decomposed into components $\mathcal{P} = (\Pi, \mathcal{V} \cup \overleftarrow{\mathcal{X}})$ and $\mathcal{O} = (\Psi, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$. Decomposition of $\mathcal{N}$ into components $\mathcal{P}$ and $\mathcal{O}$ is called* consistent *with respect to $\mathcal{N}$ iff $\mathcal{N} = \mathcal{P} \parallel \mathcal{O}$, i.e., we must have $\Sigma = \Pi \parallel \Psi$ and $\mathcal{Z} = ((\mathcal{V} \cup \overleftarrow{\mathcal{X}}) \setminus (\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma)^\bullet \cup {}^\bullet((\mathcal{W} \cup \overrightarrow{\mathcal{X}}) \setminus (\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma) \cup \sigma(\mathcal{V} \cup \overleftarrow{\mathcal{X}}, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$.*

To show that a decomposition is consistent with the original system it is sufficient to show that the set of inter-component laws of the original system is equivalent to the set of inter-component laws generated by the interface relation:

Lemma 4.4.2. *Consider a network $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$. A* consistent *decomposition of $\mathcal{N}$ into components $\mathcal{P} = (\Pi, \mathcal{V} \cup \overleftarrow{X})$ and $\mathcal{O} = (\Psi, \mathcal{W} \cup \overrightarrow{X})$ with interface relation $\sigma = \{(f(\bar{v}, a), g(\bar{v}, a), a) \mid (\bar{v}, a) \in \mathcal{X}\}$ is guaranteed if $\mathcal{X} = \sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}})$, $\mathcal{A}_\mathcal{V} \cap \mathcal{A}_{\overleftarrow{\mathcal{X}}} = \emptyset$, and $\mathcal{A}_\mathcal{W} \cap \mathcal{A}_{\overrightarrow{\mathcal{X}}} = \emptyset$.*

Proof. The decomposition of $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$ is consistent iff $\Pi \parallel \Psi = \Pi \parallel \Psi$ and $\mathcal{Z} = ((\mathcal{V} \cup \overleftarrow{\mathcal{X}}) \setminus (\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma)^\bullet \cup {}^\bullet((\mathcal{W} \cup \overrightarrow{\mathcal{X}}) \setminus (\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma) \cup \sigma(\mathcal{V} \cup \overleftarrow{\mathcal{X}}, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$. The former is trivial. Before we continue with the latter let us number the antecedent propositions of the lemma: $\mathcal{X} = \sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}})$ (1), $\mathcal{A}_\mathcal{V} \cap \mathcal{A}_{\overleftarrow{\mathcal{X}}} = \emptyset$ (2), and $\mathcal{A}_\mathcal{W} \cap \mathcal{A}_{\overrightarrow{\mathcal{X}}} = \emptyset$ (3). We will show that $\mathcal{V}^\bullet = ((\mathcal{V} \cup \overleftarrow{\mathcal{X}}) \setminus (\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma)^\bullet$, ${}^\bullet\mathcal{W} = {}^\bullet((\mathcal{W} \cup \overrightarrow{\mathcal{X}}) \setminus (\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma)$, and $\mathcal{X} = \sigma(\mathcal{V} \cup \overleftarrow{\mathcal{X}}, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$.

By construction of $\overleftarrow{\mathcal{X}}$ and definition of $\mathcal{V}_\sigma$, we have $\mathcal{A}_{\overleftarrow{\mathcal{X}}} = \mathcal{A}_{(\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma}$ and $\mathcal{A}_{\overrightarrow{\mathcal{X}}} = \mathcal{A}_{(\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma}$ (4). Furthermore, from (2) and (3) it follows that $\mathcal{V}$ and $\mathcal{W}$ are disjoint from $\overleftarrow{\mathcal{X}}$ and $\overrightarrow{\mathcal{X}}$, respectively. Thus, $\mathcal{V}$ and $\mathcal{W}$ are disjoint from $(\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma$ and $(\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma$ (5), respectively, implying that $\overleftarrow{\mathcal{X}} = (\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma$ and $\overrightarrow{\mathcal{X}} = (\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma$ (6). It follows that $\mathcal{V}^\bullet \overset{(5,6)}{=} ((\mathcal{V} \cup \overleftarrow{\mathcal{X}}) \setminus \overleftarrow{\mathcal{X}})^\bullet \overset{(6)}{=} ((\mathcal{V} \cup \overleftarrow{\mathcal{X}}) \setminus (\mathcal{V} \cup \overleftarrow{\mathcal{X}})_\sigma)^\bullet$ and, symmetrically, ${}^\bullet\mathcal{W} \overset{(5,6)}{=} {}^\bullet((\mathcal{W} \cup \overrightarrow{\mathcal{X}}) \setminus (\mathcal{W} \cup \overrightarrow{\mathcal{X}})_\sigma)$.

Recall that $\mathcal{V}$ and $\mathcal{W}$ do not have any result actions in common with $\overleftarrow{\mathcal{X}}$ and $\overrightarrow{\mathcal{X}}$, respectively (2,3), and interface actions in σ are produced by the same functions f and g that are used to produce the result actions of sets $\overleftarrow{\mathcal{X}}$ and $\overrightarrow{\mathcal{X}}$, respectively. These two facts and Definition 4.3.4 (synchronisation via σ) imply that $\sigma(\mathcal{V} \cup \overleftarrow{\mathcal{X}}, \mathcal{W} \cup \overrightarrow{\mathcal{X}}) \overset{(5,6,\sigma)}{=} \sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}}) \overset{(1)}{=} \mathcal{X}$. Hence, the decomposition of $\mathcal{N}$ is consistent if $\mathcal{X} = \sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}})$ (1), $\mathcal{A}_\mathcal{V} \cap \mathcal{A}_{\overleftarrow{\mathcal{X}}} = \emptyset$ (2), and $\mathcal{A}_\mathcal{W} \cap \mathcal{A}_{\overrightarrow{\mathcal{X}}} = \emptyset$ (3). □

It is possible to derive an inconsistent decomposition as shown in Example 4.4.3.

Example 4.4.3 (Inconsistent Decomposition)**.** *Consider a set of inter-component laws* $\mathcal{X} = \{(\langle a, b\rangle, c), (\langle b, a\rangle, c)\}$*. To generate interface result actions, consider the functions* $f(\bar{v}, a) = g(\bar{v}, a) = \alpha$ *with unique result actions* α *based solely on the result action of the input law, i.e.,* $\forall(\bar{v}', a') \in \mathcal{X}.\ a' = a \Rightarrow \alpha = f(\bar{v}', a')$*. Partitioning the laws results in the sets of interface laws* $\overleftarrow{\mathcal{X}} = \{(\langle a\rangle, \gamma), (\langle b\rangle, \gamma)\}$ *and* $\overrightarrow{\mathcal{X}} = \{(\langle b\rangle, \gamma), (\langle a\rangle, \gamma)\}$*. This system implies the interface relation* $\sigma = \{(\gamma, \gamma, c)\}$*. The derived set of inter-component laws is* $\sigma(\mathcal{V}, \mathcal{W}) = \{(\langle a, a\rangle, c), (\langle a, b\rangle, c), (\langle b, a\rangle, c), (\langle b, b\rangle, c)\} \neq \mathcal{X}$*. Hence, this decomposition is not consistent with the original system.*

However, a consistent decomposition can *always* be derived as shown in Propositions 4.4.4 and 4.4.6. These propositions give functions f and g that guarantee a consistent decomposition.

The intuition behind Proposition 4.4.4 is to encode the synchronisation laws $(\bar{v} \parallel \bar{w}, a) \in \mathcal{X}$ directly in the interface relation, i.e., we create unique result actions $\alpha_{\bar{v}}$ and $\alpha_{\bar{w}}$ with $(\alpha_{\bar{v}}, \alpha_{\bar{w}}, a) \in \sigma$. This way it is explicit which interface law corresponds to which inter-component law.

Proposition 4.4.4. *Consider a network* $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$*. We define the functions producing interface result actions as* $f(\bar{v} \parallel \bar{w}, a) = \alpha_{\bar{v}}$ *and* $g(\bar{v} \parallel \bar{w}, a) = \alpha_{\bar{w}}$*, where* $\alpha_{\bar{v}} \notin \mathcal{A}_\mathcal{V} \cup \{\tau\}$ *and* $\alpha_{\bar{w}} \notin \mathcal{A}_\mathcal{W} \cup \{\tau\}$ *are unique interface result actions identified by the corresponding interface law, that is,* $\forall(\bar{v}', a) \in \mathcal{V} \cup \overleftarrow{\mathcal{X}}.\ a = \alpha_{\bar{v}} \implies \bar{v}' = \bar{v}$ *and* $\forall(\bar{w}', a) \in \mathcal{W} \cup \overrightarrow{\mathcal{X}}.\ a = \alpha_{\bar{w}} \implies \bar{w}' = \bar{w}$*. The decomposition of* $\mathcal{N}$ *into* $\mathcal{N}_\Pi = (\Pi, \mathcal{V} \cup \overleftarrow{\mathcal{X}})$ *and* $\mathcal{N}_\Psi = (\Psi, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$ *given by* f *and* g *is* consistent*.*

Proof. Functions f and g imply interface relation $\sigma = \{(\alpha_{\bar{v}}, \alpha_{\bar{w}}, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X}\}$, and sets of interface laws $\overleftarrow{\mathcal{X}} = \{(\bar{v}, \alpha_{\bar{v}}) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X}\}$ and $\overrightarrow{\mathcal{X}} = \{(\bar{w}, \alpha_{\bar{w}}) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X}\}$.

By Lemma 4.4.2, we have to show:

- $\mathcal{X} = \sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}})$: By (1) definition of $\sigma(\mathcal{V}_\sigma, \mathcal{W}_\sigma)$, and (2) construction of $\overleftarrow{\mathcal{X}}$, $\overrightarrow{\mathcal{X}}$, and σ, it follows that

$$\begin{aligned}\sigma(\overleftarrow{\mathcal{X}}, \overrightarrow{\mathcal{X}}) &\overset{(1)}{=} \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v}, \alpha_{\bar{v}}) \in \mathcal{V}_\sigma \wedge (\bar{w}, \alpha_{\bar{w}}) \in \mathcal{W}_\sigma \wedge (\alpha_{\bar{v}}, \alpha_{\bar{w}}, a) \in \sigma\} \\ &\overset{(2)}{=} \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X}\} = \mathcal{X}\end{aligned}$$

- $\mathcal{A}_\mathcal{V} \cap \mathcal{A}_{\overleftarrow{\mathcal{X}}} = \emptyset$: Since $\alpha_{\bar{v}} \notin \mathcal{A}_\mathcal{V} \cup \{\tau\}$ the proof follows.
- $\mathcal{A}_\mathcal{W} \cap \mathcal{A}_{\overrightarrow{\mathcal{X}}} = \emptyset$: Since $\alpha_{\bar{w}} \notin \mathcal{A}_\mathcal{W} \cup \{\tau\}$ the proof follows.

□

Example 4.4.5. *Consider an admissible network with the following set of rules:*

$$\{(\langle a, \bullet, a\rangle, a), (\langle a, a, \bullet\rangle, a), (\langle b, b, b\rangle, \tau), (\langle c, \bullet, c\rangle, c)\}$$

This set of rules can be decomposed along the lines of Proposition 4.4.4 as follows:

$$\begin{aligned}\mathcal{V} &= \{(\langle a, a\rangle, a)\} \\ \mathcal{W} &= \emptyset \\ \overleftarrow{\mathcal{X}} &= \{(\langle a, \bullet\rangle, \alpha_{\langle a,\bullet\rangle}), (\langle b, b\rangle, \alpha_{\langle b,b\rangle}), (\langle c, \bullet\rangle, \alpha_{\langle c,\bullet\rangle})\} \\ \overrightarrow{\mathcal{X}} &= \{(\langle a\rangle, \alpha_{\langle a\rangle}), (\langle b\rangle, \alpha_{\langle b\rangle}), (\langle c\rangle, \alpha_{\langle c\rangle})\} \\ \sigma &= \{(\alpha_{\langle a,\bullet\rangle}, \alpha_{\langle a\rangle}, a), (\alpha_{\langle b,b\rangle}, \alpha_{\langle b\rangle}, \tau), (\alpha_{\langle c,\bullet\rangle}, \alpha_{\langle c\rangle}, c)\}\end{aligned}$$

Proposition 4.4.6 proposes an alternative decomposition that is implemented in CADP's smart reduction [56]. The idea is 1) to generate only interface relation rules of the form (a, a, b), such that components always synchronise through a common label a, while 2) keeping a equal to b and thus avoiding α labels whenever possible. Rules in this simple form make the decomposition more understandable by users.

Proposition 4.4.6. *Consider a network $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$. We define the functions producing interface result actions as*

$$f(\bar{v}, a) = g(\bar{v}, a) = \begin{cases} a & \text{if } visible\_unique(a, \mathcal{V}) \\ \alpha_{(\bar{v},a)} & \text{otherwise} \end{cases}$$

where each $\alpha_{(\bar{v},a)} \notin \mathcal{A}_\mathcal{V} \cup \mathcal{A}_\mathcal{W} \cup \{\tau\}$ is a unique interface result action identified by the corresponding inter-component law, that is, $\forall (\bar{v}', a) \in \overleftarrow{\mathcal{X}} \cup \overrightarrow{\mathcal{X}}.\ a = \alpha_{\bar{v}} \implies \bar{v}' = \bar{v}$, and where $visible\_unique(a, \mathcal{V})$ is defined by the following predicate:

$$a \neq \tau \wedge \forall (\bar{v}, a), (\bar{v}', a) \in \mathcal{V}.\ \bar{v} = \bar{v}'.$$

The decomposition of $\mathcal{N}$ into $\mathcal{N}_\Pi = (\Pi, \mathcal{V} \cup \overleftarrow{\mathcal{X}})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W} \cup \overrightarrow{\mathcal{X}})$ given by f and g is consistent.

The proof of Proposition 4.4.6 is similar to the proof of Proposition 4.4.4. The most relevant difference is the presence of $(a, a, a) \in \sigma$ if $(\bar{v} \parallel \bar{w}, a) \in \mathcal{X}$ and a is unique in $\mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X}$ ($visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$} holds). In this case we must also have $\mathcal{A}_\mathcal{V} \cap \mathcal{A}_{\overleftarrow{\mathcal{X}}} = \emptyset$ and $\mathcal{A}_\mathcal{W} \cap \mathcal{A}_{\overrightarrow{\mathcal{X}}} = \emptyset$, otherwise a contradiction with $visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$ can be derived.

The decomposition of Proposition 4.4.6 implies the interface relation

$$\begin{aligned} \sigma = &\{(\alpha_{\bar{v}\parallel\bar{w}}, \alpha_{\bar{v}\parallel\bar{w}}, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge \neg visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \\ &\cup \{(a, a, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \end{aligned}$$

and sets of interface laws

$$\begin{aligned} \mathcal{V}_\sigma = &\{(\bar{v}, \alpha_{\bar{v}\parallel\bar{w}}) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge \neg visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \\ &\cup \{(\bar{v}, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \\ \mathcal{W}_\sigma = &\{(\bar{w}, \alpha_{\bar{v}\parallel\bar{w}}) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge \neg visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \\ &\cup \{(\bar{w}, a) \mid (\bar{v} \parallel \bar{w}, a) \in \mathcal{X} \wedge visible\_unique(a, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})\} \end{aligned}$$

Example 4.4.7. *Consider again the admissible network of Example 4.4.5. Its set of rules can be decomposed along the lines of Proposition 4.4.6 as follows, using the same definition of $\mathcal{V}$ and $\mathcal{W}$:*

$$\begin{aligned} \overleftarrow{\mathcal{X}} &= \{(\langle a, \bullet\rangle, \alpha_{\langle a,\bullet,a\rangle}), (\langle b, b\rangle, \alpha_{\langle b,b,b\rangle}), (\langle c, \bullet\rangle, c)\} \\ \overrightarrow{\mathcal{X}} &= \{(\langle a\rangle, \alpha_{\langle a,\bullet,a\rangle}), (\langle b\rangle, \alpha_{\langle b,b,b\rangle}), (\langle c\rangle, c)\} \\ \sigma &= \{(\alpha_{\langle a,\bullet,a\rangle}, \alpha_{\langle a,\bullet,a\rangle}, a), (\alpha_{\langle b,b,b\rangle}, \alpha_{\langle b,b,b\rangle}, \tau), (c, c, c)\} \end{aligned}$$

Preservation of Admissibility Proposition 4.4.8 shows that LTS networks resulting from the consistent decomposition of an admissible LTS network are also admissible. Hence, consistent decomposition is compatible with the compositional verification approaches presented in [81].

Proposition 4.4.8. *Consider an admissible LTS network* $\mathcal{N} = (\Pi \parallel \Psi, \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X})$ *of length* $n + m$*. If the decomposition is consistent, then the decomposed networks* $\mathcal{N}_\Pi = (\Pi, \mathcal{V} \cup \overleftarrow{X})$ *and* $\mathcal{N}_\Psi = (\Psi, \mathcal{W} \cup \overrightarrow{X})$ *are also admissible.*

Proof. We show that $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$ satisfy Definition 2.3.2:

No synchronisation and renaming of τ*'s.* Let $(\bar{v}, a) \in \mathcal{V} \cup \overleftarrow{X}$ be a synchronisation law such that $\bar{v}_i = \tau$ for some $i \in 1..n$. We distinguish two cases:

- $(\bar{v}, a) \in \overleftarrow{X}$. Since $(\bar{v}, a)$ is an interface law and the decomposition is consistent, its result action a may not be τ. However, since $\mathcal{N}$ is admissible, no renaming of τ's is allowed. By contradiction it follows that $(\bar{v}, a) \notin \overleftarrow{X}$ completing this case.
- $(\bar{v}, a) \in \mathcal{V}$. By construction, there exists a law $(\bar{v} \parallel \bullet^m, a) \in \mathcal{V}^\bullet$. Since $\mathcal{V}^\bullet \subseteq \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X}$, by admissibility of $\mathcal{N}$, we have $\forall j \in 1..n.\ \bar{v}_j \neq \bullet \implies i = j$ (no synchronisation of τ's) and $a = \tau$ (no renaming of τ's).

Hence, $\mathcal{N}_\Pi$ does not synchronise or rename τ's. The proof for $\mathcal{N}_\Psi$ is similar.

No cutting of τ*'s.* Let Π_i be a process with $i \in 1..n$ such that $\tau \in \mathcal{A}_{\Pi_i}$. Since $\mathcal{N}$ is admissible there exists a law $(\bar{v} \parallel \bar{w}, a) \in \mathcal{V}^\bullet \cup {}^\bullet\mathcal{W} \cup \mathcal{X}$ such that $(\bar{v} \parallel \bar{u})_i = \tau$. We distinguish three cases:

- $(\bar{v} \parallel \bar{w}, a) \in \mathcal{V}^\bullet$. Since $(\bar{v} \parallel \bar{w})_i = \tau$ and $i \leq n$ it follows that $\bar{v}_i = \tau$. By construction of $\mathcal{V}^\bullet$, there is a $(\bar{v}, a) \in V$ with $\bar{v}_i = \tau$.
- $(\bar{v} \parallel \bar{w}, a) \in {}^\bullet\mathcal{W}$. In this case we must have $i > n$ which contradicts our assumption: $i \in 1..n$. The proof follows by contradiction.
- $(\bar{v} \parallel \bar{w}, a) \in \mathcal{X}$. Then, $(\bar{v} \parallel \bar{w}, a)$ is an inter-component law with at least one participating process for each component. Hence, there exists a $j \in (n+1)..m$ such that $(\bar{v} \parallel \bar{w})_j \neq \bullet$. Moreover, since $\mathcal{N}$ is admissible, no synchronisation of τ's are allowed. Therefore, since $(\bar{v} \parallel \bar{w})_j \neq \bullet$, we must have $j = i$. However, this would mean $j \in 1..n$, contradicting $j \in (n+1)..m$. By contradiction the proof follows.

We conclude that $\mathcal{N}_\Pi$ does not cut τ's. The proof for $\mathcal{N}_\Psi$ is symmetrical.

All three admissibility properties hold for $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$. Hence, the networks resulting from the decomposition satisfy Definition 2.3.2. □

4.5 Associative and Commutative LTS Network Composition

In this section we create an instance of the composition operator that is commutative and associative. This operator uses an interface relation that synchronises on a common alphabet for interface actions. It is desirable for the composition operator to be both associative and commutative as then the composition order is irrelevant with respect to the resulting system. This allows users to select any composition order; depending on the situation a particular order may be better than others [56, 61].

Definition 4.5.1 (Composition with Synchronisation on a Common Alphabet)**.** *Consider LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ of size n and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ of size m. Take the interface relation $\sigma = \{(a, a, a) \mid a \in (\mathcal{A}_\mathcal{V} \cap \mathcal{A}_\mathcal{W}) \setminus \{\tau\}\}$. The* composition on a common alphabet *of $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$ is defined as $\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi = \mathcal{N}_\Pi \parallel_\sigma \mathcal{N}_\Psi$.*

Associativity The intuition behind the associativity of LTS network composition is that vector concatenation is associative and synchronisation on the common alphabet is insensitive to the order of composition. Thus, the concatenation of process vectors and synchronisation vectors enjoys the associativity property. The challenge, however, is to show that the padding $^\bullet$ and synchronisation $\sigma(\dots,\dots)$ operations support the mathematical properties needed for associativity of the composition of sets of synchronisation laws.

Given two networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ the composition of two sets of synchronisation laws $\mathcal{V} \parallel \mathcal{W}$ consists of the union of three sets: two describing independent behaviour $(\mathcal{V} \setminus \mathcal{V}_\sigma)^\bullet$ and ${}^\bullet(\mathcal{W} \setminus \mathcal{W}_\sigma)$, and one describing synchronising behaviour $\sigma(\mathcal{V}, \mathcal{W})$ (Definition 4.3.4). When three networks are composed one (inner) composition is performed before the other. The outer composition applies the operators $^\bullet$ and $\sigma(\dots,\dots)$ on a union of sets. We show how these operators distribute over set union.

Lemma 4.5.2. *Consider sets of synchronisation laws $\mathcal{V}$ and $\mathcal{W}$ with synchronisation vectors of the same size. Padding of n $\bullet$'s distributes over set union:*

$$^\bullet(\mathcal{V} \cup \mathcal{W}) = {}^\bullet\mathcal{V} \cup {}^\bullet\mathcal{W}, \text{ and } (\mathcal{V} \cup \mathcal{W})^\bullet = \mathcal{V}^\bullet \cup \mathcal{W}^\bullet.$$

Proof. The proof of $(\mathcal{V} \cup \mathcal{W})^\bullet = \mathcal{V}^\bullet \cup \mathcal{W}^\bullet$ is analog to the proof of $^\bullet(\mathcal{V} \cup \mathcal{W}) = {}^\bullet\mathcal{V} \cup {}^\bullet\mathcal{W}$. We only prove $^\bullet(\mathcal{V} \cup \mathcal{W}) = {}^\bullet\mathcal{V} \cup {}^\bullet\mathcal{W}$ here. The proof follows from (1) application of the definition of $^\bullet$ and (2) splitting of the set on $\mathcal{V} \cup \mathcal{W}$:

$$\begin{aligned}
^\bullet(\mathcal{V} \cup \mathcal{W}) &\overset{(1)}{=} \{(\bullet^n \parallel \bar{v}, a) \mid (\bar{v}, a) \in \mathcal{V} \cup \mathcal{W}\} \\
&\overset{(2)}{=} \{(\bullet^n \parallel \bar{v}, a) \mid (\bar{v}, a) \in \mathcal{V}\} \cup \{(\bullet^n \parallel \bar{v}, a) \mid (\bar{v}, a) \in \mathcal{W}\} \\
&\overset{(1)}{=} {}^\bullet\mathcal{V} \cup {}^\bullet\mathcal{W}
\end{aligned}$$

□

Lemma 4.5.3. *Consider an interface relation σ and sets of synchronisation laws $\mathcal{V}$, $\mathcal{W}$, and $\mathcal{X}$. The $\sigma(\dots,\dots)$ operation distributes over set union as follows:*

$$\sigma(\mathcal{V}, \mathcal{W} \cup \mathcal{X}) = \sigma(\mathcal{V}, \mathcal{W}) \cup \sigma(\mathcal{V}, \mathcal{X})$$

Proof. The proof follows from (1) application of the definition of $\sigma(\dots,\dots)$ and (2) splitting of the set on $\mathcal{W} \cup \mathcal{X}$:

$$\begin{aligned}
\sigma(\mathcal{V}, \mathcal{W} \cup \mathcal{X}) &\overset{(1)}{=} \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v}, \alpha) \in \mathcal{V} \wedge (\bar{w}, \beta) \in \mathcal{W} \cup \mathcal{X} \wedge (\alpha, \beta, a) \in \sigma\} \\
&\overset{(2)}{=} \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v}, \alpha) \in \mathcal{V} \wedge (\bar{w}, \beta) \in \mathcal{W} \wedge (\alpha, \beta, a) \in \sigma\} \cup \\
&\quad\ \{(\bar{v} \parallel \bar{w}, a) \mid (\bar{v}, \alpha) \in \mathcal{V} \wedge (\bar{w}, \beta) \in \mathcal{X} \wedge (\alpha, \beta, a) \in \sigma\} \\
&\overset{(1)}{=} \sigma(\mathcal{V}, \mathcal{W}) \cup \sigma(\mathcal{V}, \mathcal{X})
\end{aligned}$$

□

Now we prove that the composition of LTS networks with synchronisation on the common alphabet is associative.

Proposition 4.5.4. *Consider LTS networks $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$, $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$, and $\mathcal{N}_\Sigma = (\Sigma, \mathcal{X})$ of sizes n, m, and o, respectively. The composition of LTS networks following Definition 4.5.1 is* associative*, i.e., it holds that $(\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi) \parallel \mathcal{N}_\Sigma = \mathcal{N}_\Pi \parallel (\mathcal{N}_\Psi \parallel \mathcal{N}_\Sigma)$.*

Proof. If the networks $(\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi) \parallel \mathcal{N}_\Sigma$ and $\mathcal{N}_\Pi \parallel (\mathcal{N}_\Psi \parallel \mathcal{N}_\Sigma)$ are equivalent, then this means that their process vectors are equivalent and their sets of synchronisation laws are equivalent.

The process vectors are equivalent due to associativity of the vector concatenation operator $\parallel$:

$$\Pi \parallel (\Psi \parallel \Sigma) = (\Pi \parallel \Psi) \parallel \Sigma$$

Next, we show $\mathcal{V} \parallel (\mathcal{W} \parallel \mathcal{X}) = (\mathcal{V} \parallel \mathcal{W}) \parallel \mathcal{X}$. First, given a set of laws $\mathcal{Y}$, we will introduce an alternative notation for $\mathcal{Y} \setminus \mathcal{Y}_\sigma$ in the context of composition $\mathcal{Y} \parallel \mathcal{Z}$ of $\mathcal{Y}$ with a set of laws $\mathcal{Z}$. We will write $\mathcal{Y} \setminus \mathcal{Z}_\mathcal{A}$ to emphasize the relevance of the alphabet of $\mathcal{Y}$ that is in common with that of $\mathcal{Z}$. We define $\mathcal{Y} \setminus \mathcal{Z}_\mathcal{A} = \{(\bar{y}, a) \in \mathcal{Y} \mid a \notin \mathcal{A}_\mathcal{Z}\}$. The set $\mathcal{Y} \setminus \mathcal{Z}_\mathcal{A}$ is equivalent to $\mathcal{Y} \setminus \mathcal{Y}_\sigma$:

$$\mathcal{Y} \setminus \mathcal{Z}_\mathcal{A} = \{(\bar{y}, a) \in \mathcal{Y} \mid a \notin \mathcal{A}_\mathcal{Z}\} = \mathcal{Y} \setminus \{(\bar{y}, a) \in \mathcal{Y} \mid (a, a, a) \in \sigma\} = \mathcal{Y} \setminus \mathcal{Y}_\sigma$$

The set $\mathcal{Z} \setminus \mathcal{Y}_\mathcal{A}$ is defined similarly.

The associativity proof proceeds as follows. Following Definition 4.3.4, we break the set $\mathcal{V} \parallel (\mathcal{W} \parallel \mathcal{X})$ down to seven partitions, and then show that there is a one-to-one mapping of these partitions to one of the seven partitions of $(\mathcal{V} \parallel \mathcal{W}) \parallel \mathcal{X}$. Each of the rewrite equations consists of four steps:

(1) unfold the outer definition of $^\bullet$ or $\sigma(\dots, \dots)$

(2) unfold the inner definition of $^\bullet$ or $\sigma(\dots, \dots)$

(3) apply associativity of vector concatenation and the inner definition of $^\bullet$ or $\sigma(\dots, \dots)$

(4) apply the outer definition of $^\bullet$ or $\sigma(\dots, \dots)$

Furthermore, in cases $2a$ and $3c$ the following property of composition of sets of laws is applied in steps (2) and (3) respectively: $\mathcal{A}_{\mathcal{W}\parallel\mathcal{X}} = \mathcal{A}_\mathcal{W} \cup \mathcal{A}_\mathcal{X}$, and hence, $a \notin \mathcal{A}_{\mathcal{W}\parallel\mathcal{X}} = a \notin \mathcal{A}_\mathcal{W} \wedge a \notin \mathcal{A}_\mathcal{X}$.

The partitioning and partition mapping proceed as follows.

1. $\sigma(\mathcal{V}, \mathcal{W} \parallel \mathcal{X})$ is partitioned, according to Lemma 4.5.3, into:

   (a) $\sigma(\mathcal{V}, \sigma(\mathcal{W}, \mathcal{X}))$, the set of laws specifying synchronisations involving all networks.

$$\begin{aligned}
&\sigma(\mathcal{V}, \sigma(\mathcal{W}, \mathcal{X})) \\
&\quad \overset{(1)}{=} \{\bar{v} \parallel (\bar{w} \parallel \bar{x}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bar{w} \parallel \bar{x}, a) \in \sigma(\mathcal{W}, \mathcal{X})\} \\
&\quad \overset{(2)}{=} \{\bar{v} \parallel (\bar{w} \parallel \bar{x}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bar{w}, a) \in \mathcal{W} \wedge (\bar{x}, a) \in \mathcal{X}\} \\
&\quad \overset{(3)}{=} \{(\bar{v} \parallel \bar{w}) \parallel \bar{x} \mid (\bar{v} \parallel \bar{w}, a) \in \sigma(\mathcal{V}, \mathcal{W}) \wedge (\bar{x}, a) \in \mathcal{X}\} \\
&\quad \overset{(4)}{=} \sigma(\sigma(\mathcal{V}, \mathcal{W}), \mathcal{X}))
\end{aligned}$$

(b) $\sigma(\mathcal{V}, (\mathcal{W} \setminus \mathcal{X}_{\mathcal{A}})^{\bullet})$, the set of laws synchronising only $\mathcal{N}_{\Pi}$ and $\mathcal{N}_{\Psi}$.

$$\begin{aligned}
&\sigma(\mathcal{V}, (\mathcal{W} \setminus \mathcal{X}_{\mathcal{A}})^{\bullet}) \\
&\quad \overset{(1)}{=} \{\bar{v} \parallel (\bar{w} \parallel \bullet^{o}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bar{w} \parallel \bullet^{o}, a) \in (\mathcal{W} \setminus \mathcal{X}_{\mathcal{A}})^{\bullet}\} \\
&\quad \overset{(2)}{=} \{\bar{v} \parallel (\bar{w} \parallel \bullet^{o}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bar{w}, a) \in \mathcal{W} \wedge a \notin \mathcal{A}_{\mathcal{X}}\} \\
&\quad \overset{(3)}{=} \{(\bar{v} \parallel \bar{w}) \parallel \bullet^{o} \mid (\bar{v} \parallel \bar{w}, a) \in \sigma(\mathcal{V}, \mathcal{W}) \wedge a \notin \mathcal{A}_{\mathcal{X}}\} \\
&\quad \overset{(4)}{=} (\sigma(\mathcal{V}, \mathcal{W}) \setminus \mathcal{X}_{\mathcal{A}})^{\bullet}
\end{aligned}$$

(c) $\sigma(\mathcal{V}, {}^{\bullet}(\mathcal{X} \setminus \mathcal{W}_{\mathcal{A}}))$, the set of laws synchronising only $\mathcal{N}_{\Pi}$ and $\mathcal{N}_{\Sigma}$.

$$\begin{aligned}
&\sigma(\mathcal{V}, {}^{\bullet}(\mathcal{X} \setminus \mathcal{W}_{\mathcal{A}})) \\
&\quad \overset{(1)}{=} \{\bar{v} \parallel (\bullet^{m} \parallel \bar{x}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bullet^{m} \parallel \bar{x}, a) \in \mathcal{X} \setminus \mathcal{W}_{\mathcal{A}}\} \\
&\quad \overset{(2)}{=} \{\bar{v} \parallel (\bullet^{m} \parallel \bar{x}) \mid (\bar{v}, a) \in \mathcal{V} \wedge (\bar{x}, a) \in \mathcal{X} \wedge a \notin \mathcal{A}_{\mathcal{W}}\} \\
&\quad \overset{(3)}{=} \{(\bar{v} \parallel \bullet^{m}) \parallel \bar{x} \mid (\bar{v} \parallel \bullet^{m}) \in (\mathcal{V} \setminus \mathcal{W}_{\mathcal{A}})^{\bullet} \wedge (\bar{x}, a) \in \mathcal{X}\} \\
&\quad \overset{(4)}{=} \sigma((\mathcal{V} \setminus \mathcal{W}_{\mathcal{A}})^{\bullet}, \mathcal{X})
\end{aligned}$$

2. $(\mathcal{V} \setminus (\mathcal{W} \parallel \mathcal{X})_{\mathcal{A}})^{\bullet}$ requires no partitioning:

(a) $(\mathcal{V} \setminus (\mathcal{W} \parallel \mathcal{X})_{\mathcal{A}})^{\bullet}$, the set of laws specifying the independent behaviour of $\mathcal{N}_{\Pi}$.

$$\begin{aligned}
&(\mathcal{V} \setminus (\mathcal{W} \parallel \mathcal{X})_{\mathcal{A}})^{\bullet} \\
&\quad \overset{(1)}{=} \{\bar{v} \parallel \bullet^{m+o} \mid (\bar{v}, a) \in \mathcal{V} \wedge a \notin \mathcal{A}_{\mathcal{W} \parallel \mathcal{X}}\} \\
&\quad \overset{(2)}{=} \{\bar{v} \parallel (\bullet^{m} \parallel \bullet^{o}) \mid (\bar{v}, a) \in \mathcal{V} \wedge a \notin \mathcal{A}_{\mathcal{W}} \wedge a \notin \mathcal{A}_{\mathcal{X}}\} \\
&\quad \overset{(3)}{=} \{(\bar{v} \parallel \bullet^{m}) \parallel \bullet^{o} \mid (\bar{v} \parallel \bullet^{m}, a) \in (\mathcal{V} \setminus \mathcal{W}_{\mathcal{A}})^{\bullet} \wedge a \notin \mathcal{A}_{\mathcal{X}}\} \\
&\quad \overset{(4)}{=} ((\mathcal{V} \setminus \mathcal{W}_{\mathcal{A}})^{\bullet} \setminus \mathcal{X}_{\mathcal{A}})^{\bullet}
\end{aligned}$$

3. ${}^{\bullet}((\mathcal{W} \parallel \mathcal{X}) \setminus \mathcal{V}_{\mathcal{A}})$ is partitioned, applying Lemma 4.5.2, into:

(a) ${}^{\bullet}(\sigma(\mathcal{W}, \mathcal{X}) \setminus \mathcal{V}_{\mathcal{A}})$, the set of laws synchronising only $\mathcal{N}_{\Psi}$ and $\mathcal{N}_{\Sigma}$.

$$\begin{aligned}
&{}^{\bullet}(\sigma(\mathcal{W}, \mathcal{X}) \setminus \mathcal{V}_{\mathcal{A}}) \\
&\quad \overset{(1)}{=} \{(\bullet^{n} \parallel (\bar{w} \parallel \bar{x}), a) \mid (\bar{w} \parallel \bar{x}, a) \in \sigma(\mathcal{W}, \mathcal{X}) \wedge a \notin \mathcal{A}_{\mathcal{V}}\} \\
&\quad \overset{(2)}{=} \{(\bullet^{n} \parallel (\bar{w} \parallel \bar{x}), a) \mid (\bar{w}, a) \in \mathcal{W} \wedge (\bar{x}, a) \in \mathcal{X} \wedge a \notin \mathcal{A}_{\mathcal{V}}\} \\
&\quad \overset{(3)}{=} \{((\bullet^{n} \parallel \bar{w}) \parallel \bar{x}, a) \mid (\bullet^{n} \parallel \bar{w}, a) \in {}^{\bullet}(\mathcal{W} \setminus \mathcal{V}_{\mathcal{A}}) \wedge (\bar{x}, a) \in \mathcal{X}\} \\
&\quad \overset{(4)}{=} \sigma({}^{\bullet}(\mathcal{W} \setminus \mathcal{V}_{\mathcal{A}}), \mathcal{X})
\end{aligned}$$

(b) $^\bullet((\mathcal{W} \setminus \mathcal{X}_\mathcal{A})^\bullet \setminus \mathcal{V}_\mathcal{A})$, the set of laws regarding independent behaviour of $\mathcal{N}_\Psi$.

$$
\begin{aligned}
&{}^\bullet((\mathcal{W} \setminus \mathcal{X}_\mathcal{A})^\bullet \setminus \mathcal{V}_\mathcal{A}) \\
&\overset{(1)}{=} \{(\bullet^n \parallel (\bar{w} \parallel \bullet^o), a) \mid (\bar{w} \parallel \bullet^o, a) \in (\mathcal{W} \setminus \mathcal{X}_\mathcal{A})^\bullet \wedge a \notin \mathcal{A}_\mathcal{V}\} \\
&\overset{(2)}{=} \{(\bullet^n \parallel (\bar{w} \parallel \bullet^o), a) \mid (\bar{w}, a) \in \mathcal{W} \wedge a \notin \mathcal{A}_\mathcal{V} \wedge a \notin \mathcal{A}_\mathcal{X}\} \\
&\overset{(3)}{=} \{((\bullet^n \parallel \bar{w}) \parallel \bullet^o, a) \mid (\bullet^n \parallel \bar{w}, a) \in {}^\bullet(\mathcal{W} \setminus \mathcal{V}_\mathcal{A}) \wedge a \notin \mathcal{A}_\mathcal{X}\} \\
&\overset{(4)}{=} ({}^\bullet(\mathcal{W} \setminus \mathcal{V}_\mathcal{A}) \setminus \mathcal{X}_\mathcal{A})^\bullet
\end{aligned}
$$

(c) $^\bullet({}^\bullet(\mathcal{X} \setminus \mathcal{W}_\mathcal{A}) \setminus \mathcal{V}_\mathcal{A})$, the set of laws specifying independent behaviour of $\mathcal{N}_\Sigma$.

$$
\begin{aligned}
&{}^\bullet({}^\bullet(\mathcal{X} \setminus \mathcal{W}_\mathcal{A}) \setminus \mathcal{V}_\mathcal{A}) \\
&\overset{(1)}{=} \{(\bullet^n \parallel (\bullet^m \parallel \bar{x}), a) \mid (\bullet^m \parallel \bar{x}, a) \in {}^\bullet(\mathcal{X} \setminus \mathcal{W}_\mathcal{A}) \wedge a \notin \mathcal{A}_\mathcal{V}\} \\
&\overset{(2)}{=} \{(\bullet^n \parallel (\bullet^m \parallel \bar{x}), a) \mid (\bar{x}, a) \in \mathcal{X} \wedge a \notin \mathcal{A}_\mathcal{W} \wedge a \notin \mathcal{A}_\mathcal{V}\} \\
&\overset{(3)}{=} \{(\bullet^{n+m} \parallel \bar{x}, a) \mid (\bar{x}, a) \in \mathcal{X} \wedge a \notin \mathcal{A}_{\mathcal{V} \parallel \mathcal{W}}\} \\
&\overset{(4)}{=} {}^\bullet(\mathcal{X} \setminus (\mathcal{V} \parallel \mathcal{W})_\mathcal{A})
\end{aligned}
$$

These equations constitute a one-to-one mapping between the partitioning of $\mathcal{V} \parallel (\mathcal{W} \parallel \mathcal{X})$ and that of $(\mathcal{V} \parallel \mathcal{W}) \parallel \mathcal{X}$. Therefore, we have $\mathcal{V} \parallel (\mathcal{W} \parallel \mathcal{X}) = (\mathcal{V} \parallel \mathcal{W}) \parallel \mathcal{X}$.

Since both $\Pi \parallel (\Psi \parallel \Sigma) = (\Pi \parallel \Psi) \parallel \Sigma$ and $\mathcal{V} \parallel (\mathcal{W} \parallel \mathcal{X}) = (\mathcal{V} \parallel \mathcal{W}) \parallel \mathcal{X}$ it follows that $(\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi) \parallel \mathcal{N}_\Sigma = \mathcal{N}_\Pi \parallel (\mathcal{N}_\Psi \parallel \mathcal{N}_\Sigma)$. □

Commutativity It is clear that composition of LTS Networks is not commutative as is indicated by Example 4.5.5.

Example 4.5.5. *Let $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ be two LTS networks. Furthermore, consider compositions $\mathcal{N}_1 = \mathcal{N}_\Pi \parallel \mathcal{N}_\Psi$ and $\mathcal{N}_2 = \mathcal{N}_\Psi \parallel \mathcal{N}_\Pi$. The network $\mathcal{N}_1$ has process vector $\Pi \parallel \Psi$ while $\mathcal{N}_2$ has process vector $\Psi \parallel \Pi$. Unless $\mathcal{N}_\Pi = \mathcal{N}_\Psi$, $\mathcal{N}_1$ and $\mathcal{N}_2$ are strictly not equivalent. Similarly, the synchronisation laws of both composite networks are in a different order.*

Network composition as defined in Definition 4.5.1 is, however, commutative with respect to the system LTS of the composition. That is, for the global behaviour of the composition of the networks, it does not matter in which order the networks are composed. We first prove that this network composition is commutative with respect to (strong) bisimulation [16] in Proposition 4.5.6. Afterwards, we will propose an adaptation of the definition of LTS network, fixing the ordering issue by replacing vectors with indexed families gaining a commutative operator for composition of LTS networks with synchronisation on the common alphabet.

Proposition 4.5.6. *Let $\mathcal{N}_\Pi = (\Pi, \mathcal{V})$ and $\mathcal{N}_\Psi = (\Psi, \mathcal{W})$ be LTS networks of sizes n and m, respectively. Composition of LTS networks according to Definition 4.5.1 is* commutative with respect to (strong) bisimulation*, i.e., it holds that $\mathcal{G}_{\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi} \leftrightarrows \mathcal{G}_{\mathcal{N}_\Psi \parallel \mathcal{N}_\Pi}$.*

Proof. Take the relation $C = \{(\bar{s} \parallel \bar{t}, \bar{t} \parallel \bar{s}) \mid \bar{s} \in \mathcal{G}_{\mathcal{N}_\Pi} \wedge \bar{t} \in \mathcal{G}_{\mathcal{N}_\Psi}\}$. The relation C is a (strong) bisimulation relation.

- *C relates the initial states of $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$.* Since every state $\bar{s} \parallel \bar{t} \in \mathcal{I}_{\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi}$ is related by C to state $\bar{t} \parallel \bar{s} \in \mathcal{I}_{\mathcal{N}_\Psi \parallel \mathcal{N}_\Pi}$ and vice versa.

- *If $\bar{s} \parallel \bar{t}\ C\ \bar{t} \parallel \bar{s}$ and $\bar{s} \parallel \bar{t} \xrightarrow{a}_{\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi} \bar{s}' \parallel \bar{t}'$ then $\bar{t} \parallel \bar{s} \xrightarrow{a}_{\mathcal{N}_\Psi \parallel \mathcal{N}_\Pi} \bar{t}'' \parallel \bar{s}'' \wedge \bar{s}' \parallel \bar{t}'\ C\ \bar{t}'' \parallel \bar{s}''$.* Let $(\bar{v} \parallel \bar{w}, a) \in \mathcal{V} \parallel \mathcal{W}$ be the law enabling the transition $\bar{s} \parallel \bar{t} \xrightarrow{a}_{\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi} \bar{s}' \parallel \bar{t}'$. It follows that there is a law $(\bar{w} \parallel \bar{v}, a) \in \mathcal{W} \parallel \mathcal{V}$ that enables the transition $\bar{t} \parallel \bar{s} \xrightarrow{a}_{\mathcal{N}_\Psi \parallel \mathcal{N}_\Pi} \bar{t}' \parallel \bar{s}'$. As $\bar{s}' \parallel \bar{t}'\ C\ \bar{t}' \parallel \bar{s}'$, the proof follows by taking $\bar{t}'$ for $\bar{t}''$, and $\bar{s}'$ for $\bar{s}''$.

- *If $\bar{s} \parallel \bar{t}\ C\ \bar{t} \parallel \bar{s}$ and $\bar{t} \parallel \bar{s} \xrightarrow{a}_{\mathcal{N}_\Psi \parallel \mathcal{N}_\Pi} \bar{t}' \parallel \bar{s}'$ then $\bar{s} \parallel \bar{t} \xrightarrow{a}_{\mathcal{N}_\Pi \parallel \mathcal{N}_\Psi} \bar{s}'' \parallel \bar{t}'' \wedge \bar{s}'' \parallel \bar{t}''\ C\ \bar{t}' \parallel \bar{s}'$.* This case is symmetric to the previous case.

□

Commutativity of composition of LTS networks To avoid the issues discussed in Example 4.5.5 an alternative definition of LTS Network can be designed. Both process vectors and synchronisation vectors may be replaced by indexed families. An *indexed family* consists of a set of objects (the process LTSs or synchronisation vectors) and an index set and a surjective function mapping elements from the index set to elements of the set of objects. When the index sets of two networks are disjoint, then the union of sets can be applied, where we previously would use vector concatenation, to compose the collections of process LTSs and synchronisation laws. The union of two indexed families is commutative, as such, commutativity of composition of LTS networks with indexed families is also commutative.

4.6 Congruence for LTS networks

In this section we prove that DPBB is a congruence for LTS Networks as defined in Definition 2.3.4 [81].

That DPBB is a congruence for LTS networks follows from associativity and commutativity rules presented in Section 4.5 for LTS networks whose set of synchronisation laws implements synchronisation on the common alphabet. That is, if for an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of some size n and all its laws $\bar{v} \in \mathcal{V}$ with some result action a it holds that each element $\bar{v}_i$ $(i \in 1..n)$ in the synchronisation vector either synchronises by firing an a-action $(\bar{v}_i = a)$, or the corresponding process has no a-transitions $(a \notin \mathcal{A}_i)$ and does not participate $(\bar{v}_i = \bullet)$ in synchronisation, then DPBB is a congruence for $\mathcal{N}$.

It is unnecessary to require that the set of synchronisation laws implements synchronisation on the common alphabet. In fact, this requirement excludes many LTS networks in practice. Therefore, we will discuss an alternative proof, presented below for Proposition 4.6.1, that does not require synchronisation on a common alphabet.

Proposition 4.6.1. *Consider two vectors of LTSs Π and Ψ, and a set of synchronisation laws $\mathcal{V}$. Furthermore, assume that τ-transitions are not renamed, cut, or synchronised in $\mathcal{V}$. It holds that*

$$(\forall i \in 1..n.\ \Pi_i \leftrightarrow_b^\Delta \Psi_i) \implies \mathcal{G}_{(\Pi,\mathcal{V})} \leftrightarrow_b^\Delta \mathcal{G}_{(\Psi,\mathcal{V})}$$

Proof. Given two vectors of Labelled Transitions Systems (LTSs) Π and Ψ such that for all $i \in 1..n$ there is a DPBB relation B_i with $\Pi_i \; B_i \; \Psi_i$, we define the bisimulation relation C as follows:

$$C = \{(\bar{s}, \bar{t}) \mid \bar{s} \in \mathcal{S}_{(\Pi,\mathcal{V})} \wedge \bar{t} \in \mathcal{S}_{(\Psi,\mathcal{V})} \wedge \forall i \in 1..n.\ \bar{s}_i \; B_i \; \bar{t}_i\}$$

We prove that C is a DPBB relation as defined in Definition 2.2.3. We will use $Ac(\bar{v}) = \{i \mid i \in 1..n \wedge \bar{v}_i \neq \bullet\}$ as a shorthand for the set of indices of processes participating in a synchronisation law $(\bar{v}, a)$; e.g., $Ac(\langle c, b, \bullet\rangle) = \{1, 2\}$.

- *C relates the initial states of $\mathcal{N}_\Pi$ and $\mathcal{N}_\Psi$.* Consider states $\bar{s} \in \mathcal{I}_{(\Pi,\mathcal{V})}$. For each $i \in 1..n$ there is a state $q_i \in \mathcal{I}_{\Psi_i}$ such that $\bar{s}_i \; B_i \; t_i$. Let $\bar{t}$ be the state built from these q_i such that $\bar{t}_i = q_i$ for all $i \in 1..n$. Then, $\bar{t} \in \mathcal{I}_{(\Psi,\mathcal{V})}$ and $\bar{s} \; C \; \bar{t}$. The symmetric case follows similarly.

- *If $\bar{s} \; C \; \bar{t}$ and $\bar{s} \xrightarrow{a}_{(\Pi,\mathcal{V})} \bar{s}'$ then either $a = \tau \wedge \bar{s}' \; C \; \bar{t}$, or $\bar{t} \xrightarrow{\tau}{}^*_{(\Psi,\mathcal{V})} \hat{\bar{t}} \xrightarrow{a}_{(\Psi,\mathcal{V})} \bar{t}' \wedge \bar{s} \; C \; \hat{\bar{t}} \wedge \bar{s}' \; C \; \bar{t}'$.* Consider a law $(\bar{v}, a) \in \mathcal{V}$ enabling transition $\bar{s} \xrightarrow{a}_{(\Pi,\mathcal{V})} \bar{s}'$. We distinguish two cases:

  1. There is a τ-action in synchronisation vector $\bar{v}$, i.e., $\exists i \in 1..n.\ \bar{v}_i = \tau$. Therefore, there is a transition $\bar{s}_i \xrightarrow{\tau}_i \bar{s}'_i$. Since τ-transitions do not synchronise it follows that it is the only action in the synchronisation vector, i.e., $\{i\} = Ac(\bar{v})$. Furthermore, $a = \tau$ as renaming τ-transitions is not allowed. Hence, by Definition 2.3.3, for all $j \in 1..n \setminus \{i\}$ it holds that $\bar{s}_j = \bar{s}'_j$. Moreover, as we also have $\bar{s}_j \; B_j \; \bar{t}_j$, it follows that $\bar{s}'_j \; B_j \; \bar{t}_j$.

     Because $\bar{s}_i \; B_i \; \bar{t}_i$ and $\bar{s}_i \xrightarrow{\tau}_i \bar{s}'_i$, by Definition 2.2.3, two cases can occur:

     - $a = \tau$ with $\bar{s}'_i \; B_i \; \bar{t}_i$. Hence, for all $j \in 1..n$ we have $\bar{s}_j \; B_j \; \bar{t}_j$. By definition of C, it follows that $\bar{s}' \; C \; \bar{t}$.
     - $\bar{t}_i \xrightarrow{\tau}{}^*_i \hat{t} \xrightarrow{a}_i t'$ with $\bar{s}_i \; B_i \; \hat{t}$ and $\bar{s}'_i \; B_i \; t'$. Since no τ-transitions are cut, there also exists a path $\bar{t} \xrightarrow{\tau}{}^*_{(\Psi,\mathcal{V})} \hat{\bar{t}} \xrightarrow{a}_{(\Psi,\mathcal{V})} \bar{t}'$ with $\hat{\bar{t}}_i = \hat{t}$, $\bar{t}'_i = t'$, and for all $j \in 1..n \setminus \{i\}$ we have $\bar{t}'_j = \hat{\bar{t}}_j = \bar{t}_j$. Therefore, from $\bar{s}_i \; B_i \; \hat{t}$, $\bar{s}'_i \; B_i \; t'$, and $\forall j \in 1..n \setminus \{i\}.\ \bar{s}_j \; B_j \; \bar{t}_j$ we deduce that $\bar{s} \; C \; \hat{\bar{t}}$ and $\bar{s}' \; C \; \bar{t}'$.

  2. There is *no* τ-action in synchronisation vector $\bar{v}$, i.e., $\forall i \in 1..n.\ \bar{v}_i \neq \tau$. By Definition 2.3.3, for all $j \in 1..n \setminus Ac(\bar{v})$ we have $\bar{s}'_j = \bar{s}_j$. Thus, since $\bar{s}_j \; B_j \; \bar{t}_j$ it follows that $\bar{s}'_j \; B_j \; \bar{t}_j$. Furthermore, we have for all $j \in Ac(\bar{v})$ a transition $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$. Hence, as $\bar{v}_j \neq \tau$ for all those $j \in Ac(\bar{v})$, there exists a path $\bar{t}_j \xrightarrow{\tau}{}^*_j \hat{\bar{t}}_j \xrightarrow{\bar{v}_j} \bar{t}'_j$ with $\bar{s}_j \; B_j \; \hat{\bar{t}}_j$ and $\bar{s}'_j \; B_j \; \bar{t}'_j$ (by Definition 2.2.3). From Definition 2.3.3 it follows that there also is a path $\bar{t} \xrightarrow{\tau}{}^*_{(\Psi,\mathcal{V})} \hat{\bar{t}} \xrightarrow{a}_{(\Psi,\mathcal{V})} \bar{t}'$ where for all $j \in 1..n \setminus Ac(\bar{v})$ $\hat{\bar{t}}_j$ and $\bar{t}'_j$ are defined by $\bar{t}'_j = \hat{\bar{t}}_j = \bar{t}_j$. Hence, from $\forall i \in 1..n.\ \bar{s}_i \; B_i \; \bar{t}_i$ and $\forall j \in Ac(\bar{v}).\ \bar{s}_j \; B_j \; \hat{\bar{t}}_j$, and $\forall k \in Ac(\bar{v}).\ \bar{s}'_k \; B_k \; \bar{t}'_k$ we deduce that $\bar{s} \; C \; \hat{\bar{t}}$ and $\bar{s}' \; C \; \hat{\bar{t}}'$.

- *If $\bar{s} \; C \; \bar{t}$ and $\bar{t} \xrightarrow{a}_{(\Psi,\mathcal{V})} \bar{t}'$ then either $a = \tau \wedge \bar{s}' \; C \; \bar{t}$, or $\bar{s} \xrightarrow{\tau}{}^*_{(\Pi,\mathcal{V})} \hat{\bar{s}} \xrightarrow{a}_{(\Pi,\mathcal{V})} \bar{s}' \wedge \bar{s} \; C \; \hat{\bar{t}} \wedge \bar{s}' \; C \; \bar{t}'$.* This case is symmetric to the previous case.

- *If $\bar{s} \; C \; \bar{t}$ and there is an infinite sequence of states $(\bar{s}^k)_{k\in\omega}$ such that $\bar{s} = \bar{s}^0$, $\bar{s}^k \xrightarrow{\tau}_{(\Pi,\mathcal{V})} \bar{s}^{k+1}$ and $\bar{s}^k \; C \; \bar{t}$ for all $k \in \omega$, then there exists a state $\bar{t}'$ such that*

$\bar{t} \xrightarrow{\tau}{}^{+}_{(\Psi,\mathcal{V})} \bar{t}'$ *and* $\bar{s}^k \; C \; \bar{t}'$ *for some* $k \in \omega$. For all $k \in \omega$ let $\bar{v}^k$ be the synchronisation law enabling transition $s^k \xrightarrow{\tau} s^{k+1}$.

We distinguish two cases:

- There is a $k \in \omega$ such that $\bar{s}^k \xrightarrow{\tau} \bar{s}^{k+1}$ is the result of the synchronisation of multiple processes in Π, i.e., $\exists k \in \omega, i \in 1..n.\ \{i\} \subset Ac(\bar{v}^k)$. In the τ-sequence $\bar{s}^\ell \; C \; \bar{t}$ for all $\ell \in \omega$, hence, we have $\bar{s}^k \; C \; \bar{t}$. Furthermore, since C is a bisimulation relation it follows that there are states $\hat{\bar{t}}, \bar{t}' \in \mathcal{S}_{(\Psi,\mathcal{V})}$ with a τ-path $\bar{t} \xrightarrow{\tau}{}^{*}_{(\Psi,\mathcal{V})} \hat{\bar{t}} \xrightarrow{\tau}_{(\Psi,\mathcal{V})} \bar{t}'$ such that $\bar{s}^{k+1} \; B \; \bar{t}'$. Thus, $\bar{t} \xrightarrow{\tau}{}^{+}_{(\Psi,\mathcal{V})} \bar{t}'$ and for $k+1 \in \omega$ it holds that $\bar{s}^{k+1} \; C \; \bar{t}'$ completing the case.
- The τ-sequence only consists of τ-transitions performed independently by the processes in Π, i.e., $\forall k \in \omega.\ |Ac(\bar{v}^k)| = 1$. Since all of the τ-transitions are performed independently, there has to be at least one process of which its infinite τ-sequence is embedded in the global infinite τ-sequence, otherwise the given τ-sequence starting from s would not be infinite. Suppose the i^{th} process has such a sequence and that the sequence starts from state $\bar{s}_i$. The independent infinite sequence is embedded in that of the network, hence, for all $k \in \omega$ it holds that $\bar{s}_i^k \; B_i \; \bar{t}_i$. Since $\bar{s}_i \; B_i \; \bar{t}_i$, by Definition 2.2.3, there is a state $t' \in \mathcal{S}_{\Psi_i}$ with $\bar{t}_i \xrightarrow{\tau}{}^{+}_{\Psi_i} t'$ and some $\ell \in \omega$ such that $\bar{s}_i^\ell \; B_i \; t'$. We construct state $\bar{t}'$ such that for all $j \in 1..n$ if $j = i$, then $\bar{t}'_j = t'$, and otherwise $\bar{t}'_j = \bar{t}_j$. As local τ-transitions are not cut nor renamed, it follows that $\bar{t} \xrightarrow{\tau}{}^{+}_{(\Psi,\mathcal{V})} \bar{t}'$. Moreover, since $\bar{s}^k \; C \; \bar{t}$ for all $k \in \omega$, by definition of C, we have $\bar{s}_j^\ell \; B_j \; \bar{t}_j$ for all $j \in 1..n$. Finally, because $\bar{s}_i^\ell \; B_i \; \bar{t}'_i$ and for all $j \in 1..n \setminus \{i\}$ it holds that $\bar{s}_j^\ell \; B_j \; \bar{t}_j$, by construction of $\bar{t}'$ it follows that $\bar{s}^\ell \; C \; \bar{t}'$.

- *If* $\bar{s} \; C \; \bar{t}$ *and there is an infinite sequence of states* $(\bar{t}^k)_{k\in\omega}$ *such that* $\bar{t} = \bar{t}^0$, $\bar{t}^k \xrightarrow{\tau}_{(\Psi,\mathcal{V})} \bar{t}^{k+1}$ *and* $\bar{s} \; C \; \bar{t}^k$ *for all* $k \in \omega$, *then there exists a state* $\bar{s}'$ *such that* $\bar{s} \xrightarrow{\tau}{}^{+}_{(\Pi,\mathcal{V})} \bar{s}'$ *and* $\bar{s}' \; C \; \bar{t}^k$ *for some* $k \in \omega$. This case is symmetric to the previous case.

□

4.7 Application

In order to compare compositional approaches with the classical, non-compositional approach, we have employed CADP to minimise a set of test cases modulo DPBB.

Each test case consists of a model that is minimised with respect to a given liveness property. To achieve the best minimisation we applied maximal hiding [144] in all approaches. Intuitively, maximal hiding hides all actions except for the interface actions and actions relevant for the given liveness property.

As *composition strategy* we have used the *smart reduction* approach described in [56]. In CADP, the *classical approach*, where the full state space is constructed at once and no intermediate minimisations are applied, is the *root reduction* strategy. At the start, the individual components are minimised before they are combined in parallel composition, hence the name.

We have measured the *running time* and the *maximum number of states and transitions* generated by the two methods.

Experimental setup To facilitate replication we briefly discuss the methods used for our experiments.

For compositional approaches, the running time and largest state space considered depends heavily on the composition order, i.e., the order in which the components are combined. The smart reduction approach uses a heuristic to determine the order in which to compose processes. In [56], it has been experimentally established that this heuristic frequently works very well. After each composition step the result is minimised.

We use the following expression from the SVL scripting language [80] of CADP to invoke the smart reduction modulo DPBB approach:

```
smart total divbranching reduction of (<m>)
```

where <m> is the test model.

In the classical approach the state space of the entire system is generated before minimisation is applied. This approach is invoked as follows

```
root total divbranching reduction of (<m>)
```

where <m> is the test model.

The experiments were run on the DAS-5 cluster [17] machines. They have an INTEL HASWELL E5-2630-v3 2.4 GHz CPU, 64 GB memory, and run CENTOS LINUX 7.2. The running time of the two approaches was measured as the wall clock time (i.e., the real elapsed time) using the Unix time command:

```
/usr/bin/time -f "%e" svl <file>
```

The argument `-f "%e"` specifies that the time written as output should follow format `"%e"` where `%e` indicates the wall clock time. The `svl <script>` argument invokes the SVL-engine with script `<script>`. The command measures the wall clock time of the execution of the SVL-script.

The maximum number of states and transitions that were generated were extracted from the SVL-log files after execution of the script.

The set of test cases As test input we selected 19 case studies: four MCRL2 [54] models distributed with its tool set, nine CADP models, three from the BEEM database [162], and three from Example Repository for Finite State Verification Tools [130].

The models stemming from the MCRL2 tool set distribution are the following:

1. The *1394* model, created by Luttik [141], specifies the 1394 or firewire protocol. *Property*: every PAreq with parameter immediate is eventually followed by a matching PAcon with parameter won.

2. The *1394'* model is the 1394 model scaled up with extra internal transitions. This model is our own adaptation of the 1394 model and is therefore not distributed with the tool set. *Property*: same as the 1394 model.

3. The *ACS* model describes the ACS Manager that is part ALMA project of the European Southern Observatory. The ACS Manager is part of a system controlling a large collection of radio telescopes. The model consists of a manager and some containers and components and was created by Ploeger [169]. *Property*: every time container MT1 is locked, eventually it is freed again.

4. *Wafer Stepper* models a wafer stepper used in the manufacturing of integrated circuits. *Property*: always, eventually, all wafers in the system will be exposed.

The CADP models consist of:

1. *Cache* models a directory-based cache coherency protocol for a multi-processor architecture. The model was developed by Kahlouche et al.[111]. *Property*: there is no live-lock.

2. The *DES* model describes an implementation of the data encryption standard, which allows to cipher and decipher 64-bit vectors using a 64-bit key vector [153]. *Property*: the DES can always deliver outputs.

3. *HAVi-LE* describes the asynchronous Leader Election protocol used in the HAVi (Home Audio-Video) standard, involving three device control managers. The model is fully described by Romijn [177]. *Property*: always eventually a leader is selected.

4. *HAVi-LE'* is an adaptation of the HAVi-LE model containing transitions denoting logging events. Since the model is our own adaptation it is not distributed with CADP. *Property*: same as the HAVi-LE model.

5. *Le Lann* models a distributed leader election algorithm for unidirectional ring networks. The CADP model was developed by Garavel and Mounier [83]. *Property*: process `P0` is infinitely many times in the critical section.

6. *ODP* is a model of an open distributed processing trader [84]. *Property*: work is always executed eventually.

7. The *Eratosthenes Sieve* model computes prime numbers implementing a distributed Eratosthenes Sieve; the model describes a pipeline of units, of which each unit blocks input numbers that are multiples of a given number. The model consists of four units. *Property*: if the number two is generated, then it is eventually reported as a prime number.

8. *Eratosthenes Sieve'* is a variant of Eratosthenes Sieve consisting of seven units. *Property*: same as the Eratosthenes Sieve model.

9. The *Transit* model describes a transit-node, it models an abstraction of a routing component of a communication network. The model was developed by Mounier [151]. *Property*: every time a message is receive, it is eventually either sent out the node or buffered as faulty.

The BEEM models are:

1. The *Peterson* model describes Peterson's mutual exclusion algorithm [167] for seven processes. *Property*: every time process `P0` waits for access to the critical section, it will eventually enter it.

2. *Anderson* models Anderson's queue lock mutual exclusion algorithm [10] for three processes. *Property*: Every time a process waits for access to the critical section, it will eventually enter it.

3. *Anderson'* is a variant of the Anderson model considering four processes competing for a lock. *Property*: same as the Anderson model.

Table 4.1: Experiments: smart reduction vs. root reduction; running times are presented in seconds

Test case	Running time		Maximum #states		Maximum #transitions	
	smart	root	smart	root	smart	root
1394	14.41	**8.25**	**102,983**	198,692	**187,714**	355,338
1394'	**47.51**	460.53	**2,832,074**	36,855,184	**5,578,078**	96,553,318
ACS	70.87	**11.22**	**1,854**	4,764	**4,760**	14,760
Anderson	26.56	**15.42**	**153,664**	384,104	**2,118,368**	5,892,964
Anderson'	**373.56**	1852.42	**15,116,544**	56,250,000	**268,738,560**	1,188,000,000
Cache	20,55	**7.84**	616	616	4,631	4,631
Chiron	22.76	**13.66**	**317,115**	481,140	**2,563,650**	3,456,675
Chiron'	**1,171.06**	1,236.06	**49,076,280**	56,293,380	**467,536,860**	513,857,520
DES	**54.61**	948.66	**1,404**	64,498,297	**3,510**	518,438,860
Eratosthenes Sieve	63.00	**8.43**	1,156,781	**234**	2,891,692	**406**
Eratosthenes Sieve'	–	**10.64**	–	**865**	–	**2,012**
Gas Station	**325.10**	362.31	**11,042,816**	11,436,032	**84,254,720**	87,105,536
HAVi-LE	**114.27**	493.01	**970,772**	15,688,570	**5,803,552**	80,686,289
HAVi-LE'	**93.08**	5,255.56	**453,124**	190,208,728	**2,534,371**	876,008,628
Le Lann	**96.35**	5,599.15	**12,083**	160,025,986	**701,916**	944,322,648
ODP	32.90	**9.97**	**10,397**	91,394	**87,936**	641,226
Peterson	**63.04**	–	**9**	–	**139**	–
Transit	**25.50**	59.69	**22,928**	3,763,192	**132,712**	39,925,524
Wafer Stepper	74.18	**57.54**	**962,122**	3,772,753	**4,537,240**	16,977,692

From the Example Repository for Finite State Verification Tools we selected the following models:

1. The *Chiron* model describes a user interface development system with two clients. The system consists of the Chiron server, managing generic aspects of a user interface, and artists (the clients). This server is responsible for notifying artists when a user interface event occurs, while the clients listen for notifications from the server. The formal model was developed by Avrunin et al. [15]. *Property*: If an artist is registered for event `e1`, then it will eventually be notified for this event.

2. *Chiron'* is a adapted version of the Chiron model where another client is added. There are a total of three clients. *Property*: same as the Chiron model.

3. The *Gas Station* problem [97] simulates a self-serve gas station. The gas station consists of two pumps, an operator, and three customers. *Property*: A charge is made eventually after a customer has started pumping.

Measurement Results The results of our experiments are shown in Table 4.1. The *Test case* column indicates the test case model corresponding to the measurements.

The *smart* and *root* sub-columns denote the measurement for the smart reduction and root reduction approaches, respectively.

In the *Running time* column the running time until completion of the experiment is shown in seconds. Indicated in bold are the shortest running times comparing the *smart* and *root* sub-columns. The maximum running time of an experiment was set to 80 hours, after which the experiment was discontinued (indicated with –).

The columns *Maximum #states* and *Maximum #transitions* show the largest number of states and transitions, respectively, generated during the experiment. Of both methods the best result is indicated in bold.

Discussion In terms of *running time* smart reduction performs best for 10 out of 19 models, whereas root reduction performs best in 8 of the models. In terms of both *maximum number of states* and *maximum number of transitions* smart reduction outperforms root reduction in 16 out of 19 models.

In general, the smart reduction approach performs better for large models where the state space can be reduced significantly before composition. This is best seen in the *HAVi-LE'*, *Le Lann*, and *Peterson* test cases, where smart reduction is several hours faster.

In this set of models, root reduction performs best in relatively small models; *1394*, *ACS*, *Cache*, *Lamport*, and *ODP*. However, for these cases the difference in running times is negligible. Smart reduction starts performing better in the moderately sized models such as *Transit* and *Wafer stepper*. For smaller models the overhead of the smart reduction heuristic is too high to obtain any benefits from the nominated ordering.

Smart reduction performs particularly bad for the *Eratosthenes Sieve'* model. The model consists of a pipeline where data is being pushed from one end to another. While the data domain considered by the nodes in the pipeline consists of 32 elements, in the minimised state space only one element remains. As synchronising actions may not be hidden in the local process LTSs the incremental composition and minimisation leads to a state space that is several orders of magnitude larger than the final state space.

In summary, compositional reduction is most efficient when it is expected that components reduce significantly and highly interleaving components are added last.

4.8 Conclusions

In this chapter, we have shown that DPBB is a congruence for parallel composition of LTS networks where there is synchronisation on given label combinations. Therefore, the DPBB equivalence may be used to reduce components in the compositional verification of LTS networks. As DPBB is the finest equivalence relation in the linear time - branching time spectrum this allows reduction of the state space while preserving a larger class of properties than other relations in this spectrum.

Furthermore, we have discussed how to safely decompose an LTS network in the case where verification has to start from the system as a whole. Both the composition and consistent decomposition of LTS networks preserve the admissibility property of LTS networks. Hence, the composition operator remains compatible with the compositional verification approaches for LTS networks described by [81].

We have shown that parallel composition of LTS networks with synchronisation on the common result action alphabet is associative and commutative. From this it follows that DPBB is also a congruence for LTS networks as defined by Garavel, Lang, and Mateescu [81] if the set of synchronisation laws implements synchronisation on the common alphabet. We have shown, however, that the requirement to synchronise on the common alphabet is unnecessarily restrictive. This has been shown in a direct proof of DPBB being a congruence for LTS networks.

The proofs in this chapter have been mechanically verified (with exception Proposition 4.6.1) using the Coq proof assistant [3] and are available online. [4] The mechanical

[3] `https://coq.inria.fr`.

[4] `http://www.win.tue.nl/mdse/composition/DPBB_is_a_congruence_for_synchronizing_LTSs.zip`

verification of the related proofs in this chapter gives us confidence in the correctness of Proposition 4.6.1.

Although our work focuses on the composition of LTS networks, the results are also applicable to composition of individual LTSs. Our parallel composition operator subsumes the usual parallel composition operators of standard process algebra languages such as CCS [148], CSP [179], mCRL2 [54], and LOTOS [106].

Finally, we have run a set of experiments to compare compositional and traditional DPBB reduction. The compositional approach applies Cadp's smart reduction employing a heuristic to determine an efficient compositional reduction order. The traditional reduction generates the complete state space before applying reduction. The compositional approach performed better when applied to the medium to large models where the intermediate state space can be kept small.

Future Work An interesting direction for future work is the integration of the proof in a meta-theory for process algebra. This integration would give a straightforward extension of our results to parallel composition for process algebra formalisms.

This work has been inspired by an approach for the compositional verification of transformations of LTS networks [59, 214, 215, 217, 218]. We would like to apply the results of this chapter to the improved transformation verification algorithm [59], thus guaranteeing its correctness for the compositional verification of transformations of LTS networks.

In future experiments, we would like to involve recent advancements in the computation of branching bisimulation, and therefore also DPBB, both sequentially [91] and in parallel on graphics processors [216]. It will be interesting to measure the effect of applying these new algorithms to compositionally solve a model checking problem.

Finally, by encoding timing in the LTSs, it is possible to reason about timed system behaviour. Combining approaches such as [213, 219] with our results would allow to compositionally reason about timed behaviour. We plan to investigate this further.

Chapter 5

To Compose, Or Not to Compose: An Analysis of Compositional Minimisation

To tackle the state space explosion problem several compositional verification approaches have been proposed. One of these approaches is compositional aggregation, where a given system consisting of a number of parallel processes is iteratively composed and minimised. Compositional aggregation has shown to perform better (in the size of the largest state space in memory at one time) than classical monolithic composition in a number of cases. However, there are also cases in which compositional aggregation performs significantly worse. It is unclear when one should apply compositional aggregation in favour of other techniques and how it is affected by action hiding and the scale of the model.

This chapter presents a descriptive analysis following the quantitative experimental approach. The experiments were conducted in a controlled test bed setup in a computer laboratory environment. A total of eight scalable models with different network topologies considering a number of varying properties were investigated comprising 119 subjects. This makes it the most comprehensive study done so far on the topic. We investigate whether there is any systematic difference in the success of compositional aggregation based on the model, scaling, and action hiding. Our results indicate that both scaling up the model and hiding more behaviour has a positive influence on compositional aggregation.

Correlation, regression, and classification analysis was conducted on 1,615 generated networks of LTSs for a number of potential predictors. Maximum memory cost and maximum number of generated transitions of heuristics normalised with respect the monolithic approach was predicted at accuracy of about one order of magnitude. Classifying the best minimisation approach achieved an accuracy between 55% and 61% for maximum memory cost, and between 68% and 74% for the maximum number of generated transitions.

This chapter is an extension of

[61] de Putter, S., and Wijs, A. To Compose, or Not to Compose, That Is the Question: An Analysis of Compositional State Space Generation. In *FM* (2018), Springer International Publishing, pp. 485–504

5.1 Introduction

Although model checking [16] is one of the most successful approaches for the analysis and verification of the behaviour of concurrent systems, it is plagued with the so-called *state space explosion problem*: the state space of a concurrent system tends to increase exponentially as the number of parallel processes increases linearly.

To tackle the state space explosion several compositional approaches have been proposed such as assume-guarantee reasoning [110, 171] and partial model checking [8]. Cobleigh, Avrunin, and Clarke conducted an evaluation of assume-guarantee reasoning in 2008 [49]. They raises doubt whether it is an effective alternative to classical, monolithic model checking.

A prominent alternative approach is compositional aggregation [55, 74] (also known as compositional state space generation [203], incremental composition and reduction [183], incremental reachability analysis [196, 197], and inductive compression [180]). Given a system consisting of a number of parallel processes the *compositional aggregation* approach iteratively composes the processes and minimises the result. *Action abstraction* or *hiding* [144] may be applied to abstract away all actions irrelevant for the property being verified such that minimisation is more effective. The idea of compositional aggregation is that incremental minimisation should warrant a lower maximum memory cost than composing the system monolithically. Compositional aggregation has shown to perform better (in the size of the largest state space in memory) than monolithic composition in a number of cases [55, 56, 64, 81, 196]. However, sometimes the former is not effective, even producing a (significantly) larger state space than the monolithic approach [81].

The aggregation order of a composition can be understood as a tree, where leaves are the parallel processes and the nodes represent an operation that constructs a composite Labelled Transition System (LTS) from the children nodes and minimises the result. As such the number of possible aggregation orders is exponential in the number of parallel processes. The selection of an efficient order, i.e., that results in compositional aggregation being as memory efficient as possible is still an unsolved issue [55].

To automate the selection of the aggregation order several heuristics have been proposed [55, 56, 196]. However, it is unpredictable whether aggregation orders selected by the heuristics are an improvement over the monolithic approach. Insights in the conditions in which compositional aggregation is expected to perform well are vital for successful application of the techniques, but these insights are currently limited. Evaluation of compositional aggregation and heuristics is, to the best of our knowledge, only limited to small benchmarks with no control on aggregation order, model scale, and action hiding. To gain understanding on how these variables influence the effectiveness of compositional aggregation, this chapter presents a *characterisation* of the compositional aggregation method. The *objective* of this study is as follows:

> Analyse *compositional aggregation* for the purpose of *characterisation* of the *maximum memory use of the generated state space* in the context of *aggregation orderings of concurrent models with different scaling and action hiding.*

The goal is to find guidelines that help deciding whether to apply compositional aggregation. To this end we address the following main research question.

> **RQ main:** *When can compositional aggregation be expected to be more (memory) efficient than monolithic minimisation?*

To answer this question we first answer a number of smaller questions. First, we investigate the effect of three specific aspects of the application of compositional aggregation: the aggregation order, the amount of action hiding, and the number of parallel processes in the model that compositional aggregation is applied to.

> **RQ 5.1:** *How do action hiding, number of parallel processes, and aggregation order affect the memory consumption of compositional aggregation?*

As stated earlier, some aggregation orders are better than others. Heuristics are employed in an attempt to find the well performing aggregation orders. Therefore, to determine whether or not it is wise to apply compositional aggregation the performance of the heuristics must be kept in mind.

> **RQ 5.2:** *How effective are the aggregation orders chosen by current heuristics?*

Having established what minimisation approach is most efficient on which variants of the models, we finally investigate the relation between subjects within these two groups (compositional aggregation and monolithic minimisation). Answering this research question provides insights into which structural properties of models are indicative for the success or failure of compositional aggregation.

> **RQ 5.3:** *How can the success or failure of compositional aggregation be explained?*

Finally, to answer our main research question we investigate the predictive power of several properties and learned predictors for the success or failure of compositional aggregation.

> **RQ 5.4:** *How can the success or failure of compositional aggregation be predicted?*

In terms of scaling, due to the exponential growth of aggregation orders, we limit the number of analysed aggregation orders to 2,500. The action hiding sets are derived from properties formulated for the corresponding models using the *maximal hiding* technique [144]. Finally, for minimisation we use branching bisimulation with explicit divergence [204] as it supports a broad range of safety and liveness properties.

Contributions We present our findings after having conducted a thorough experiment to study the effectiveness of compositional aggregation when applied to models with varying network topologies.

Having analysed a significant number of possible aggregation orders, we were able to compare several heuristics proposed in the literature with (near-)optimal composition results. In total, we have selected 119 subjects for the analysis, making this the most comprehensive study performed on the topic so far. Our main conclusion is that the amount of internal behaviour of individual processes in the model, and the amount of synchronisation between those processes, seem to be the two main factors influencing the success of compositional aggregation. Furthermore, our results suggest that there is real potential to construct better heuristics in the near future.

To develop prediction models that assist in selection of a minimisation approach, we have performed correlation tests and applied regression, and classification algorithms on 1,615 networks of LTSs using a number of potential predictors. These networks were

generated pseudo-randomly based on 88 source process LTSs, since we do not have a sufficient number of LTS networks available to give enough statistical power. The analysis shows that the maximum memory cost of compositional aggregation heuristics normalised with respect to monolithic minimisation can be predicted at an average accuracy of about one order of magnitude. The best learned classification models determining the best minimisation approach with respect to maximum memory cost and maximum number of generated transitions were validated against new data. The models classified between 55% and 61% of the unseen data correctly; telling whether monolithic minimisation or smart reduction performs best in terms of memory. For the maximum number of generated transitions metric the classification models had an accuracy between 68% and 74% on unseen data. Furthermore, as both the training data and test data contain a class imbalance the Cohen's κ measure for inter-rate agreement is between 0.10 and 0.22, indicating that the models are better than random classification, but there is room for improvement.

The prediction models could be further improved through inclusion of relevant variables, more data, and the introduction of regression and classification techniques that are specialised for data in the form of labelled graphs (such as LTSs). Metrics related to interleaving density and the number of transitions in an LTS were most important for regression techniques, while metrics related to hiding and interleaving density of sets of LTSs where found to be the most determining factor classification techniques.

Note that the study was conducted on networks of LTSs and, therefore, the results are possibly limited to models represented as networks of LTSs.

Structure of the chapter In Section 5.2, we discuss related work. Preliminaries are given in Section 5.3. The methodology used in our experiment is discussed in Section 5.4. Section 5.5 presents our results for RQ 5.1, RQ 5.2, and RQ 5.3. Special attention is given to RQ 5.4 in Section 5.6. Finally, conclusions and future work are discussed in Section 5.8.

5.2 Related Work

Compositional aggregation. In the past, compositional aggregation has been applied in a number of experiments [56, 81, 197]. Tai and Koppol [197] do consider a small set of orders for each considered case, and they target a set of models mostly consisting of randomly generated models and variations of only one or two real use cases. Considering a small set of orders makes it hard to indicate the quality of the considered heuristics, i.e., how well they perform compared to how well they *could* potentially perform. The usefulness of insights gained by analysing randomly generated models heavily depends on how similar the models are to real models, in terms of their structural characteristics.

Crouzen and Lang [56] developed two of the three heuristics proposed by Tai and Koppol further and combined this into what the authors call *smart reduction*. Crouzen and Lang consider a benchmark set of 28 models that are variants of 13 models. This is a relatively high number of subjects, but unfortunately, discussion of the results is very limited, and the differences between subjects based on the same model are not explained. Due to this, the effect of these differences between the subjects cannot be correlated to the presented performance.

Garavel, Lang, and Mateescu [81] subject the combined heuristic to another experiment to show the effect of action hiding, i.e., abstraction of behaviour irrelevant for the

considered functional property. The experiment measures the largest number of states generated during aggregation with and without action hiding. The experiment considers 90 subjects; a single (industrial) use case consisting of 5 scenarios, each considering a subset of 25 properties. They report that action hiding improves the performance of the heuristic. It is not reported whether there is a correlation between the amount of reduction and the properties.

Graf and Steffen [90] were the first to propose context constraints for compositional aggregation. A method for automatically generating context constraints for compositional aggregation methods is proposed by Cheung and Kramer [40]. It consists of generating an interface LTS representing the communicating behaviour of a set of components, and then composing this interface with the remainder of the components. The resulting state space is weakly bisimilar to the monolithically generated state space. To evaluate the approach the authors perform several experiments with client/server models that are scaled by adding clients to the model. In each experiment the aggregation order was fixed. In contrast, we both scale the models and vary the aggregation orders to see how they affect the effectiveness of the technique.

Other compositional approaches. An evaluation of automated assume-guarantee reasoning was conducted by Cobleigh, Avrunin, and Clarke [49]. The authors study whether assume-guarantee reasoning provides an advantage over monolithic verification. They conclude by raising doubts whether assume-guarantee reasoning is an effective compositional verification approach. However, no attempts were made to investigate the effects of combining multiple components in one step, i.e., n-way decomposition, and action hiding. Assume-guarantee reasoning may be more effective when these approaches are involved.

Assume-guarantee reasoning by abstraction refinement [85] improves upon the approach. The technique is inspired by the experience that small interfaces between components positively affect compositional reasoning. The study considers four cases with a total of twelve subjects. The improved approach uses less memory than the original one in seven of the twelve subjects. However, it is not reported how the memory consumption is measured (i.e., of what the memory consumption is measured exactly), and furthermore, the results are not compared to monolithic verification.

An n-way decomposition with alphabet refinement is proposed by Abd Elkader et al. [2]. A benchmark consisting of three cases with a total of fifteen subjects is performed, but memory consumption is not reported. In eight of the fifteen subjects, the approach turned out to be faster than monolithic verification.

Other contributions to assume-guarantee reasoning [96, 161] present similarly small benchmarks with the number of cases not exceeding four and the number of subjects not exceeding seventeen. Gupta, McMillan, and Fu [96] report the memory consumption as the number of states in an assumption LTS, however, no correlation with actual memory consumption is discussed. Păsăreanu et al. [161] report the memory consumption of the used tools. Still, all these benchmarks compare few behavioural models that are verified with respect to different properties. However, all variants are treated as independent subjects, thus skewing results.

Concluding, compared to our study, none of the related studies consider (non-random) models of varying network topologies, and take those topologies explicitly into account. We also study in detail the effect of action hiding. Furthermore, in none of the studies the results are corrected for repeated measures, which occur when you obtain results from variations of test cases. Finally, it should be noted that most studies consider not enough

cases and subjects to extract general conclusions.

5.3 Background

Our experiments are performed using CADP [81]. In this section, we explain the computational model behind the compositional aggregation technique offered by CADP. An LTS (Definition 2.2.1) describes the behaviour of a process or system. The behaviour of a concurrent system is described by a *network of LTSs* [133] (Definition 2.3.1), or *LTS network* for short. From an LTS network, a *system LTS* (Definition 2.3.3) can be derived describing the global behaviour of the network. The state space of these systems may be minimised modulo an appropriate equivalence relation.

The *minimisation* of an LTS consists of the merging of all states that have equivalent behaviour. A new, reduced, LTS is obtained that is equivalent to the original LTS. An equivalence relation R between two LTSs relates states that have equivalent behaviour. We write $min_R(\mathcal{G})$ for the minimisation of an LTS $\mathcal{G}$ according to an equivalence relation R. Given an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$, we write $min_R(\mathcal{N})$ and $\min_R(\Pi, \mathcal{V})$ as a short hand for $min_R(\mathcal{G}_\mathcal{N})$ and $min_R(\mathcal{G}_{(\Pi,\mathcal{V})})$, respectively, denoting the minimisation of the system LTS of a network modulo an equivalence relation R. The minimisation of an LTS network consists of iterative composition and minimisation of its process LTSs in some order until a complete minimised system LTS is obtained. When minimising an LTS network, actions of processes that require synchronisation with those of other processes cannot be abstracted away prematurely.

In this chapter we use the DPBB equivalence relation [204] (Definition 2.2.3) as relation fo minimisation. DPBB supports action hiding and is sensitive to branching structure and cycles of τ-transitions, i.e., infinite internal behaviour.

To maximise the potential for minimisation, *maximal hiding* [144] can be applied, which identifies exactly which actions are essential to correctly determine whether an LTS satisfies a given functional property or not. This roughly corresponds to hiding all actions except those occurring in the formula. Maximal hiding is defined over a fragment of the modal μ-calculus that is adequate with respect to DPBB. The defined fragment is expressive enough to express most properties.

Projection of LTS networks To aggregate processes of a network, these processes must be selected from the network. To this end, we define the projection of a network $\mathcal{N}$ on to a set of indices I. The result is an LTS network that can be considered a subsystem or component of $\mathcal{N}$ consisting of the processes originally indexed in Π at the positions indicated by I.

Definition 5.3.1 (Projection of an LTS network)**.** *Consider a set of indices I and an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$. Let $\alpha(\bar{v}, a) \notin \mathcal{A}_\mathcal{V}$ be a function associating a unique action label to each $(\bar{v}, a) \in \mathcal{V}$. The projection of $\mathcal{N}$ on to I is defined as $\mathcal{N}^I = (\Pi^I, \mathcal{V}^I)$ where Π^I is the vector project of Π on to I and*

$$\begin{aligned} \mathcal{V}^I \;=\; & \{\ (\bar{v}^I, a) \qquad\quad \mid (\bar{v}, a) \in \mathcal{V} \wedge Ac(\bar{v}) \subseteq I \qquad\quad \} \cup \\ & \{\ (\bar{v}^I, \alpha(\bar{v}, a)) \mid (\bar{v}, a) \in \mathcal{V} \wedge \emptyset \subset (I \cap Ac(\bar{v})) \subset Ac(\bar{v})\}. \end{aligned}$$

The synchronisation laws of the projected network $\mathcal{N}^I$ are divided in two disjoint subsets:

- The *first* subset represents synchronisation laws that only involve process LTSs inside I. These laws do not synchronise with processes that are not in I.
- The *second* subset represents the synchronisation laws that involve process LTSs in I as well as other process LTSs (those in $I^{\complement}$, the complement of I). As these laws must still synchronise with LTSs outside I, the unique action label $\alpha(\bar{v}, a)$ is introduced to restore synchronisation with other LTSs when $\mathcal{N}^I$ is composed with processes outside I.

Compositional order. The compositional aggregation of an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ is the incremental composition and minimisation of subsets of processes in Π. More specifically, the composition of a set of LTSs followed by a minimisation of the result is called an *aggregation*. The *compositional aggregation* modulo R of an LTS network $\mathcal{N}$ is the incremental aggregation of the processes in Π subject to $\mathcal{V}$ such that the resulting LTS is R-equivalent to $\mathcal{G}_{\mathcal{N}}$. Before we formally define compositional aggregation, we must first introduce aggregation orders.

An *aggregation order* organises the processes of an LTS network in a tree-structure as presented in Definition 5.3.2. The leaves represent the individual process LTSs in Π, and the nodes represent subsets of the processes in Π. The root represents all the processes in Π. For the sake of simplicity, the processes are represented by their index in the process vector Π. Let 2^I denote the power set of a set I.

Definition 5.3.2 (Aggregation Order)**.** *Given an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of size n, an aggregation order of $\mathcal{N}$ is a tree $T_{\mathcal{N}} = (V, E)$ where $\emptyset \subset V \subset 2^{1..n}$ such that*

- *$1..n$ is the root of the tree,*
- *The singleton sets $\{i\} \in V$ with $i \in 1..n$ are the leaves of the tree, and*
- *For every non-leaf node $t \in V$, the children of t must form a partitioning of t.*

Consider a node $t \in V$. The set of values associated with node t is denoted $val(t)$. Furthermore, the list of children of t is regarded as a vector, i.e., t_i indicates the i^{th} child of node t, and $|t|$ denotes the number of children of t. Finally, in our examples within this section we represent aggregation orders by *nested sets* described by the following EBNF syntax:

```
Tree = '{' Tree | i {',' Tree | i} '}'
```

where $i \in 1..n$. For instance, we write $\{\{\{1,3\},2\},\{4,5,6\}\}$ to represent the aggregation tree shown in Figure 5.1.

The *compositional aggregation* of a network $\mathcal{N}$ according to an aggregation order T is formalized in Definition 5.3.3. Let t be the root of aggregation order tree T, compositional aggregation *first* decomposes $\mathcal{N}$ by projecting $\mathcal{N}$ *on the sets* and *by pre-order walk* of the aggregation order, as indicated by the *second case* of $agg_R(\mathcal{N}, t)$. That is, each component represented by a child of t is aggregated before finally constructing and minimising the state space. *Second*, minimisation starts at the leaves when no further decomposition is possible as indicated by the *first case* of $agg_R(\mathcal{N}, t)$. *Finally*, aggregation (state space generation and minimisation) is performed in a post-order walk through the created aggregation tree (i.e., children are processed before their parents). The children of the nodes represent the (system) LTSs of the components. At each non-leaf node t the state space of component t is constructed by concatenating the process vectors of the

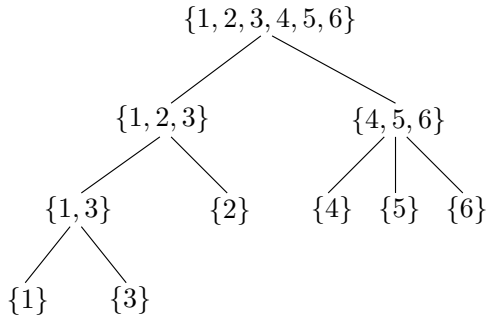


Figure 5.1: An aggregation order represented by $\{\{\{1,3\},2\},\{4,5,6\}\}$: first all leaves are reduced; next, processes $\{1,3\}$, and $\{4,5,6\}$ are aggregated; then, the aggregation of $\{1,3\}$ is aggregated with process 2; finally, the remaining two LTSs are aggregated to obtain the aggregation of the whole system

child networks and restoring synchronisations through $\mathcal{V}_{snc}$. The set $Ac(\bar{v})^t$ is the set of children (represented by index) of a node t that are involved in a synchronisation vector $\bar{v}$.

Definition 5.3.3 (Compositional Aggregation)**.** *Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network, and let T be an aggregation order with root t. The* compositional aggregation *of $\mathcal{N}$ subject to order T is defined as*

$$agg_R(\mathcal{N}, t) = \begin{cases} min_R(\Pi_i) & \text{if } val(t) = \{i\} \text{ for } i \in 1..n \\ min_R(\|_{i=1}^{|t|} agg_R(\mathcal{N}^{val(t_i)}, t_i), \mathcal{V}_{snc}) & \text{otherwise} \end{cases}$$

where

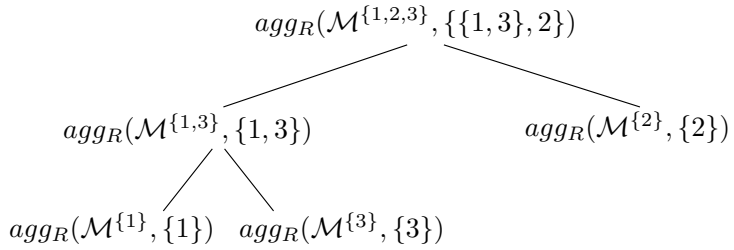
$$\begin{aligned} \mathcal{V}_{snc} = \{\, (\bullet^{|t|}[i \mapsto a], a) \quad & | \ (\bar{v}, a) \in \mathcal{V} \wedge Ac(\bar{v})^t = \{i\} \,\} \\ \cup \{\, (\bullet^{|t|}[Ac(\bar{v})^t \mapsto \alpha(\bar{v}, a)], a) \quad & | \ (\bar{v}, a) \in \mathcal{V} \wedge |Ac(\bar{v})^t| > 1 \\ & \quad \wedge Ac(\bar{v}) \subseteq val(t) \,\} \\ \cup \{\, (\bullet^{|t|}[Ac(\bar{v})^t \mapsto \alpha(\bar{v}, a)], \alpha(\bar{v}, a)) \quad & | \ (\bar{v}, a) \in \mathcal{V} \wedge Ac(\bar{v}) \cap val(t) \neq \emptyset \\ & \quad \wedge Ac(\bar{v}) \not\subseteq val(t) \,\}, \end{aligned}$$

and

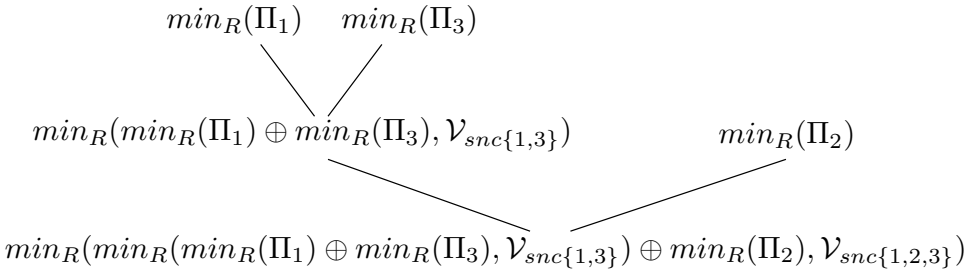
$$Ac(\bar{v})^t = \{i \mid i \in 1..|t| \wedge Ac(\bar{v}) \cap val(t_i) \neq \emptyset\}$$

The synchronisation laws $\mathcal{V}_{snc}$ of an aggregated component consist of three disjoint subsets. Given an aggregation order node t, a representative of a synchronisation law $(\bar{v}, a)$ is in the first set iff only one child of t is involved in $\bar{v}$, $(\bar{v}, a)$ is in the second set iff multiple children of t are involved in $\bar{v}$, and in the third set iff $\bar{v}$ involves LTSs that are not represented by t. The laws in the sets are formulated over the aggregated child components. That is, the synchronisation vectors are of size $|t|$ and for each $\bar{v} \in \mathcal{V}_{snc}$ and $i \in 1..|t|$, $\bar{v}_i$ is the action label performed by the aggregation of the i^{th} child network. The set $\mathcal{V}_{snc}$ is organised as follows:

- The *first* subset constrains the number of child components that participate in a law $(\bar{v}, a) \in \mathcal{V}$ to a single component: $Ac(\bar{v})^t = \{i\}$. This set contains for each child



(a) A tree representation of the aggregation calls and decompositions corresponding to the aggregation of a network $\mathcal{N}$ in pre-order walk through aggregation order $\{\{1,3\},2\}$



(b) A tree representation of aggregation of a network in post-order walk through aggregation order $\{\{1,3\},2\}$, $\mathcal{V}_{sncI}$ represents the set of synchronisation laws $\mathcal{V}_{sync}$ created at node I of the aggregation order

Figure 5.2: The compositional aggregation of a network $\mathcal{N} = (\Pi, \mathcal{V})$ according to aggregation order $\{\{1,3\},2\}$ (the left sub-tree of the aggregation order shown in Figure 5.1)

component laws that solely involve the child component. Hence, these laws allow the child components to perform their independent actions.

- The *second* subset considers the laws $(\bar{v}, a) \in V$ which involve multiple child components of t ($|Ac(\bar{v})^t| > 1$) and only child components of t ($Ac(\bar{v}) \subseteq val(t)$). The child components involved in $(\bar{v}, a)$ all synchronise on a special action $\alpha(\bar{v}, a)$ that is produced by the projection of the network $\mathcal{N}^{val(t)}$ on to the subsets $val(t_i)$ with $i \in 1..|t|$. The result action of the synchronisation of children on $\alpha(\bar{v}, a)$ is the a action of the original law $(\bar{v}, a)$.

- The *third* subset contains the laws that involve the component represented by t ($Ac(\bar{v}) \cap val(t) \neq \emptyset$), but also components *not* represented by t ($Ac(\bar{v}) \not\subseteq val(t)$). A law in this set describes synchronisation of all involved child components on the special $\alpha(\bar{v}, a)$ action. The result is again the $\alpha(\bar{v}, a)$ action indicating that the component must still synchronise with other parts of the system.

Figure 5.2 shows an example of the compositional aggregation of an LTS network consisting of three parallel processes. The aggregation order followed is $\{\{1,3\},2\}$, the left sub-tree in Figure 5.1 shows a tree representation of this aggregation order. First, in Figure 5.2a, the aggregation calls decompose the network in pre-order walk through the aggregation order until the leaves are reached; parents are decomposed before forming decomposed networks passed on to the children. Then, in Figure 5.2b, the aggregations are applied in post-order walk through the aggregation order; composition and minimisation are applied to children before their parents.

Compositional aggregation in CADP. The SVL scripting language[1] of CADP offers (amongst others) the following minimisation generation strategies:

- The monolithic approach, referred to as *root reduction*, directly computes the system LTS of an LTS network and then applies minimisation.
- *Root leaf reduction* applies minimisation to the process LTSs of a network and then applies root reduction on the resulting network.
- *Smart reduction* [56] is a heuristic that attempts to find an efficient aggregation order. First, all the process LTSs are minimised. Then, recursively, a set I of process LTSs is selected and the LTSs in I are replaced by their aggregation.

Metrics for compositional aggregation There are a number of metrics that are of interest for compositional aggregation. One of the most simple methods is to consider the number of transitions in each process LTS. The number of transitions in a process LTS $\mathcal{G}$ is denoted by $|\mathcal{T}_{\mathcal{G}}|$.

Smart reduction selects the aggregation order based on a combination of two metrics: the *Hiding Metric* (*HM*), an indication of the density of hidden transitions, and the *Interleaving Metric* (*IM*), an indication of the interleaving density. The sum of these two metrics is called the *Combined Metric* (*CM*).

Smart reduction scores the sets of processes it considers for aggregation using the *CM* metric. The processes considered are selected by a set of process indices I. The *HM* and *IM* metrics use an upper bound of the number of transitions in the aggregation of the processes in I.

Definition 5.3.4 (Upper Bound of the Number of Transitions of an Aggregation [56,81])**.** *Given an LTS network $\mathcal{N}$, an* upper bound *for the number of transitions in the aggregation of the processes indicated by a set of process indices I is defined as follows*

$$ET(I,\bar{v}) = \begin{cases} 0 & \text{if } I \cap Ac(\bar{v}) = \emptyset \\ \prod_{i \in I \setminus Ac(\bar{v})} |\mathcal{S}_i| \times \prod_{i \in I \cap Ac(\bar{v})} |\xrightarrow{\bar{v}_i}| & \text{otherwise} \end{cases}$$

Informally, $ET(I,\bar{v})$ computes, for synchronisation vector $\bar{v}$, the maximum number of transitions going out of every product state of I (this includes unreachable states).

Smart reduction's *CM* consists of *HM* and *IM*. Hiding in an aggregation is promoted via *HM*: the *IM* metric has a higher value for aggregations with high normalised hiding rate. Conversely, interleaving in the aggregation is punished via *IM* which has a lower value for aggregations with higher estimated amounts of interleaving.

HM gives an indication of the density of hidden transitions (compared to a total number of transitions) normalised by the number of processes considered for aggregation.

Definition 5.3.5 (Hiding Metric [56,81])**.** *Given a set of process indices I over a network $\mathcal{N}$, the* hiding metric *is defined by $HM(I) = HR(I)/|I|$, where HR (the hiding rate) is defined by*

$$HR(I) = \frac{\sum_{(\bar{v},\tau) \in \mathcal{V} \wedge Ac(v) \subseteq I} ET(I,v)}{1 + \sum_{(\bar{v},a) \in \mathcal{V}} ET(I,\bar{v})}$$

IM gives an indication of the density of interleaving transitions (compared to a fully connected LTS) normalised by the number of processes considered for aggregation.

[1] For more info see `http://cadp.inria.fr/man/svl-lang.html`

Definition 5.3.6 (Interleaving Metric [56, 81]). *Given a set of process indices I over a network $\mathcal{N}$, the* interleaving metric *is defined by $IM(I) = (1 - IR(I))/|I|$, where IR (the interleaving rate) is defined as*

$$IR(I) = \frac{\sum_{(\bar{v},a)\in\mathcal{V}} ET(I,v)}{1 + \sum_{(\bar{v},a)\in\mathcal{V}} \sum_{i\in I\cap Ac(\bar{v})} ET(I, \bar{v}@i)}$$

where $\bar{v}@i_j = \bar{v}_i$ if $i = j$ and $\bar{v}@i_j = \bullet$ otherwise.

CM used by smart reduction is defined as follows.

Definition 5.3.7 (Combined Metric [56, 81]). *Given a set of process indices I over a network $\mathcal{N}$, the* combined metric *is defined by $CM(I) = HM(I) + IM(I)$.*

Other metrics of interest to compositional aggregation concern the ratio of synchronising transitions resulting in τ of a set of processes. High density of such synchronisations is desirable as they do not cause interleaving and can be reduced by the aggregation. These kind of metrics are dual to a metric such as *IM*, promoting synchronisation as opposed to punishing interleaving (*IM*).

Given a set of processes indices I, the Hidden Synchronisation Density Vector (*HSDV*) is a vector containing the ratio of hidden synchronisations within I for each processes indicated by I.

Definition 5.3.8 (Hidden Synchronisation Density Vector). *Consider a network $\mathcal{N}$ and a set of process indices I over $\mathcal{N}$. The* synchronised hiding density vector *is defined as follows $HSDV(I) = \langle\ |\mathcal{T}_{\tau-sync}(\Pi_i, I)|/|\mathcal{T}_{\Pi_i}|\ |\ i \in I\rangle$ where $\mathcal{T}_{\tau-sync}(\Pi_i, I)$ are the transitions of Π_i that synchronise with the other processes indicated by I resulting in the invisible action τ.*

The vector produced by *HSDV* can be used to compute an indication of the hidden synchronisation between a set of process indices I.

Definition 5.3.9 (Weighted Hidden Synchronisation Density). *Consider a network $\mathcal{N}$ and a set of process indices I over $\mathcal{N}$. The* weighted synchronisation density *is defined by $WHSD(I) = \Pi_{e\in HSDV(I)}\ e$.*

The values of the vector of $HSDV(I)$ are multiplied to represent the common wish to synchronise. If only a few members of I have a high hidden synchronisation density, and other have a low density, then the high density members will not have much opportunities to synchronise; in this case $HSDV(I)$ will be low to indicate the lack of interest in (hidden) synchronisation. Conversely, if all members of I have a value close to 1, then $WHSD(I)$ will also be close to 1, indicating that all processes have high hidden synchronisation density.

5.4 Methodology

Setup Our experiments were conducted in a controlled test bed comprising a set of homogeneous machines from the DAS-4 [14] cluster. Each machine has a dual quad-core Intel Xeon E5620 2.4 GHz CPU, 24 GB memory, and runs CentOS Linux 6. We used Cadp version 2017-e "Sophia Antipolis" as implementation for the monolithic and compositional aggregation approaches.

The monolithic approach has been used as the control group. For compositional aggregation, all possible aggregation orders were computed using Refiner [218] in combination with the `decomposition.brute_force` plugin. The minimisation strategies were coded in the Script Verification Language (Svl) [80] of Cadp. Given a property the hiding set was calculated using the maximal hiding technique [144]. This technique produces a set of property relevant actions that may not be hidden in the system. All other actions can be safely hidden without affecting the verification result.

As *cases* we consider LTS network models in Cadp's EXP format. As *subjects* we consider case instances with a particular scale and hiding set. We use *minimisation strategy* to refer to both aggregation according to some order, and monolithic minimisation.

Research Questions The variable of interest, i.e., the response variable, is the maximum memory cost (measured as the maximum measured resident set size) the compositional aggregation method. However, due to techniques Cadp uses to compress the state space it is hard to relate this response variable to other variables. Moreover, for small cases the maximum memory cost shows little variation hindering comparison even further.

To gain more insight we also consider the *maximum number of transitions* generated as an alternative measure for maximum memory cost. The maximum number of transitions of an LTS has a medium to strong and highly significant correlation with the memory cost of minimisation in Cadp, especially for large models: according to 41 groups (subjects with a large number of parallel processes); for 26 out of 41 groups these two metrics are strongly correlated (coefficient of 0.7 or higher), and for 10 more groups they are moderately correlated (coefficient of between 0.7 and 0.5). For all of these the p-value is at most 1.5×10^{-2}. Correlations have been measured using Kendall's τ_b coefficient [117]. An additional advantage is that the maximum number of transitions metric is tool agnostic.

To answer RQ 5.1 (see Section 5.1) we measured the *maximum memory cost* and *maximum number of transitions* among the state spaces produced by compositional aggregation for *all possible* aggregation orders on a set of subjects. The effect of scaling and action hiding were investigated by controlling, respectively, the number of parallel processes and the property.

Next, *the performance of current heuristics are compared to that of other aggregation orders* in RQ 5.2. The *smart reduction* and *root leaf reduction* heuristics were applied to the subjects. Both heuristics are supported by Cadp and have shown to be competitive with respect to other heuristics [56]. Again, we measured the *maximum memory cost* and *maximum number of transitions* among the state spaces processed by compositional aggregation.

The intention of RQ 5.3 is to *explain the success or failure of composition aggregation.* Observed difference in performance between the subjects of the cases were investigated closer by inspecting the effect of action hiding, number of parallel processes, and aggregation order. Findings were verified with adjusted models fixing one or more aspects, therefore, obtaining more controlled measurements.

Finally, RQ 5.4 aims to *identify or learn predictors that predict whether heuristics for compositional aggregation will perform better or worse than monolithic minimisation.* For this, more data was needed to increase statistical significance of the analysis. As a solution 2,000 models were generated based on a number of extracted metrics of 88 process LTSs originating from 29 LTS networks. A series of LTS metrics were considered as well as graph metrics over the topology of the models. The selected metrics were tested for correlation with the response variables. Additionally, machine learning techniques,

offered by the `carat` R package [125], were used to learn predictors.

There are numerous variables that may affect the performance of compositional aggregation with respect to monolithic minimisation. Variables of interest are typically related to the size of a process LTS, or the reduction or interleaving that a process LTS or the composition of process LTSs may introduce.

5.4.1 Case and Subject Selection for RQ 5.1, RQ 5.2, and RQ 5.3

The cases were sampled using *quota sampling* [158], i.e., cases with various characteristics were selected. To avoid source bias the cases were selected from four different sources, and where needed, converted to LTS networks.

***Source* 1** The BEnchmark for Explicit Model checkers (BEEM) database [162]. The benchmark includes 57 parametric models with corresponding properties. [2]

***Source* 2** The demos of the CADP distribution. The CADP distribution contains a set of 42 demos. Many of the demos were extracted from the numerous real world verification case studies performed with CADP.

***Source* 3** The cases considered in an evaluation of automated assume-guarantee reasoning [49]. This considered set consists of 5 scalable cases with corresponding properties. [3]

***Source* 4** The cases considered in our previous work [60]. In previous work we experimented with a set of 10 cases of which some are scalable.[4]

As mentioned by Cobleigh et al. [49] the generality of their work is threatened by the limited variety in network topology. To avoid this, we selected cases with a variety of network topologies. In addition, we took the following considerations into account:

1. The effect of action hiding was considered by selecting for each case various relevant safety and liveness properties.

2. To investigate the effect of the number of parallel LTSs on compositional aggregation we selected scalable cases. Each scalable case has one or more repeatable LTSs with which the model was scaled up; e.g., a model consisting of a single server LTS and two client LTSs was scaled up by adding copies of the client LTSs.

3. The number of possible aggregation orders and the time required to construct state spaces grow exponentially with scale. Due to time considerations we limited each compositional aggregation to two hours. In addition, we prematurely terminated a compositional aggregation procedure as soon as it required more than the available (physical) memory, i.e., 24 GB. Any subjects violating the time or memory criteria were discarded from the experiment.

4. It is infeasible to calculate all the 39,208 possible aggregation orders at seven parallel LTSs within reasonable time. Therefore, we chose to limit the number of considered aggregation orders to a random sample 2,500 aggregation orders per subject. The size of the sample is large enough to form a significant representation of the 39,208 possible aggregation orders at seven parallel LTSs.

[2] `http://paradise.fi.muni.cz/beem`.
[3] `http://laser.cs.umass.edu/breakingup-examples`.
[4] `http://www.win.tue.nl/mdse/property_preservation/FAC2017_experiments.zip`.

Initially the sources above provided 115 models. We selected a number of scalable cases with a variety of network topologies. We discarded the cases for which it was infeasible to compute 2,500 aggregation orders for at least two scaled up version of the case. Finally, *eight* cases were selected covering *five* different network topologies. *Six* out of the *eight* cases were able to scale to a size of *six* parallel LTSs while satisfying the time and memory criteria. The other two cases were scaled to *four* and *seven* parallel LTSs, respectively.

Next, we selected a range of properties relevant for the cases and modelled several scaled-up LTS networks. This finally resulted in a total of 129 subjects. The experiments were run on these 129 subjects. In total 119,078 decompositions were considered costing a total of 2.4 CPU-years. Finally, for 119 subjects all the run aggregation orders satisfied the time and memory criteria.

5.4.2 Case Selection for RQ 5.4

The few cases considered for the other research questions are not nearly enough to find or learn predictors that are statistically significant. For this we need hundreds of cases, if not more. As there are no sources (that we are aware of) that add up to the number of cases required, the only solution is to generate these cases.

Pelánek [163] has shown that random graphs differ significantly from state spaces. Most notably, state spaces typically contain diamonds due to parallel composition and have different distributions of node degrees and labels.

Hence, we have avoided the *first* issue by generating LTS networks consisting of generated process LTSs. The state space of such an LTS network has the famous diamonds, but the individual LTSs do not.

The *second* problem was resolved by generating process LTSs where node degrees and transition labels are sampled from a large number of existing process LTSs from the sources mentioned in the previous section (Section 5.4.1).

We found a *third* problem with random graphs: they often contain many unreachable nodes. In contrast, a process LTS typically has no unreachable states. To combat this issue we have first generated a spanning-tree for each process LTS such that all states were reachable from one state.

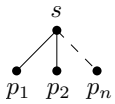
Obtaining data The distributions used for the generation of process LTSs were: *node in-degree, node out-degree, labels, and synchronising labels.* These distributions were extracted from a total of 88 process LTSs covering 29 distinct LTS networks.[5] Each *process LTS* was computed by sampling the distributions of one of the 88 source process LTSs.

Topologies of LTS networks were generated following a uniform degree distribution. Topology degree distributions were not sampled for the generation of topologies as this could have caused a lack of variation (due to the small size of the topology graph).

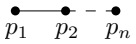
Finally, with the (unlabelled) process LTSs and the topologies ready, the process LTSs were labelled according to the label and synchronising label distributions selected for each process LTS. Synchronisation of labels was coordinated via the generated topology.

With this approach 2,000 LTS networks were generated with *five* process LTSs each. This number of process LTSs was chosen to simplify analysis: by fixing the number of process LTSs the effect it has on the predictor variable is removed.

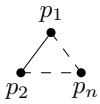
[5]The networks are available at: `http://www.win.tue.nl/mdse/composition/ML_models.zip`



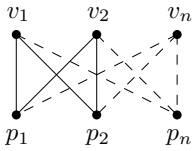
(a) Clients p_i ($i \in 1..n$) and server s



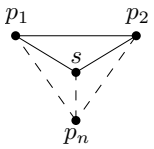
(b) Pipes and filters with processing nodes p_i ($i \in 1..n$)



(c) Ring with processing nodes p_i ($i \in 1..n$)



(d) Processes p_i sharing variables v_i ($i \in 1..n$)



(e) Peer-to-peer network with peers p_i ($i \in 1..n$) and tracker-server s

Figure 5.3: Network topologies

Smart reduction, root leaf reduction, and monolithic minimisation were applied to all the generated LTS networks. For the minimisations the no deadlock property was considered, i.e., all (global) result actions were hidden.

Model selection criteria The method in which labels are selected for transitions has a risk of generating an LTS network in which only deadlock states are reachable. We have removed these, since these networks do not represent realistic systems. Furthermore, a minimisation strategy applied to a given network was cancelled after 2 hours to limit expensive resource consumption.

Of the 2,000 generated models the minimised state space was computed within 2 hours by at least one minimisation strategy for 1,615 models.[6] All of these minimised models contained at least one transition. These will be the considered cases in the following analysis.

A brief inspection of the data indicated a clear class imbalance. Smart reduction out performed the other two approaches in over 80% of the cases. A prediction model that would always select smart reduction as the best possible minimisation method would automatically have an accuracy of around 80%. We have taken this into account in our analysis methodology.

5.5 Results – RQ 5.1, RQ 5.2, and RQ 5.3

5.5.1 Case and subject descriptions

Network topologies The selected cases are characterised by the network topologies depicted in Figure 5.3. Dots indicate parallel processes and lines indicate synchronisation relations. Dashed lines show the synchronisation relations that are introduced by adding a repeatable process p.

Figure 5.3a shows a *client-server* topology. Such a network contains one or more servers and one or more clients.

[6]The the resulting data is available at: `http://www.win.tue.nl/mdse/composition/data_gen_net.zip`

Table 5.1: Selected cases and their characteristics; with $p \geq 1$ the # of repeated LTSs

Case ID	Case description	Topology	Scaling	Source
1	The Gas Station problem [97]	a (3 servers)	$3 + p \geq 4$	1,3
2	Chiron user interface (single dispatcher) [115]	a (2 servers)	$3 + p \geq 4$	3
3	Eratosthenes' Sieve (distributed calculation of primes)	b	$1 + p \geq 3$	2
4	Le Lann leader election protocol [138]	c	$2 \cdot p \geq 4$	1
5	A simple token ring	c	$1 + p \geq 3$	4
6	Peterson's mutual exclusion protocol [167]	d	$2 \cdot p \geq 4$	1,2,3
7	Anderson's mutual exclusion protocol [10]	d	$1 + 2 \cdot p \geq 5$	1
8	Open Distributed Processing trader (ODP) [84]	e	$1 + p \geq 3$	2

In Figure 5.3b a *pipes and filter* topology is presented. The first process p_1 produces data and each process p_i ($i \in 1..n$) in the sequence processes the data and filters before forwarding the filtered data to the next process p_{i+1}.

A *ring* network topology is shown in Figure 5.3c. Communication between processes is organised as a ring structure. Often a token is passed along the edges that grants special privileges to the process holding the token.

Figure 5.3d depicts communication via a number of *shared variables*. In the selected cases, for each repeatable process p_i there is a repeatable variable v_i.

In Figure 5.3e a *peer-to-peer* network topology is shown. Addresses and services of the peers p_i ($i \in 1..n$) are published via the tracker-server s after which the offered services can be employed on a peer-to-peer basis.

Case descriptions We have selected *eight* scalable models as cases. An overview of these cases is given in Table 5.1. We identify the cases by their case number indicated in the *Case ID* column. The *Scaling* column shows the scaling of the cases in the number of repeated LTSs p and, on the right-hand side of the inequality, the minimum number of parallel LTSs; e.g., ODP's scaling $1 + p \geq 3$ states that there is one non-repeated LTS (the trader) and one repeated LTS (the client), but the number of processes must be at least 3. Finally, in the *Source* column the sources of the cases are given, these correspond to the list of sources (Section 5.4). [7]

Subject descriptions Subjects correspond to instances of cases with a particular scale and hiding set, i.e., property. Subjects are identified by three alphanumeric characters: the first indicating the number of the case ID, the second indicating the letter of a corresponding case property, and the third indicating the scale of the case model. With "_", we denote the absence of a property, i.e., no hiding is applied. For instance, `1e5` is the case `1` model where actions not relevant to property `e` (of case `1`) have been hidden and the subject has a total of `5` parallel LTSs. For each model, we identified between two and eight relevant properties.

The selected *scaling* is from the minimum scale of the case up to the possible scale nearest to six; e.g., for case `1` with property `a` the set of subjects is `1a4`, `1a5`, `1a6` and for case `6` with no property the set of subjects is `6_4`, `6_6`.

[7] The models are available at `http://www.win.tue.nl/mdse/composition/test_cases.zip`.

5.5.2 Analysis

Figures 5.4 and 5.5 respectively show the distribution of the normalised memory consumption and normalised maximum number of transitions of the generated state spaces for all possible aggregation orders of each subject, in the form of violin plots [101].[8] The black horizontal lines within each plot connected by a black vertical line indicate the first, second, and third quartiles. On the x-axis the subjects are displayed, grouped by case ID and scale. The y-axis displays the value of the response variables on a $\log_{10}$-scale: maximum memory cost in Figure 5.4 and maximum number of transitions in the generated state space in Figure 5.5. Furthermore, the dashed horizontal line indicates the performance of monolithic construction. Finally, the normalised values of the response variables for *smart reduction* and *root leaf reduction* are indicated by a *red dot* and *blue diamond*, respectively.

It should be noted that the repeating of LTSs has a noticeable effect on the distribution of aggregation orders. Some peaks arise due to accumulation of sets of symmetric aggregation orders measuring the same normalized maximum number of transitions. However, as can be seen in the plots, in most cases this effect does not change significantly as more repeated LTSs are added.

5.5.2.1 RQ 5.1 How do action hiding, number of parallel processes, and aggregation order affect the memory consumption of compositional aggregation?

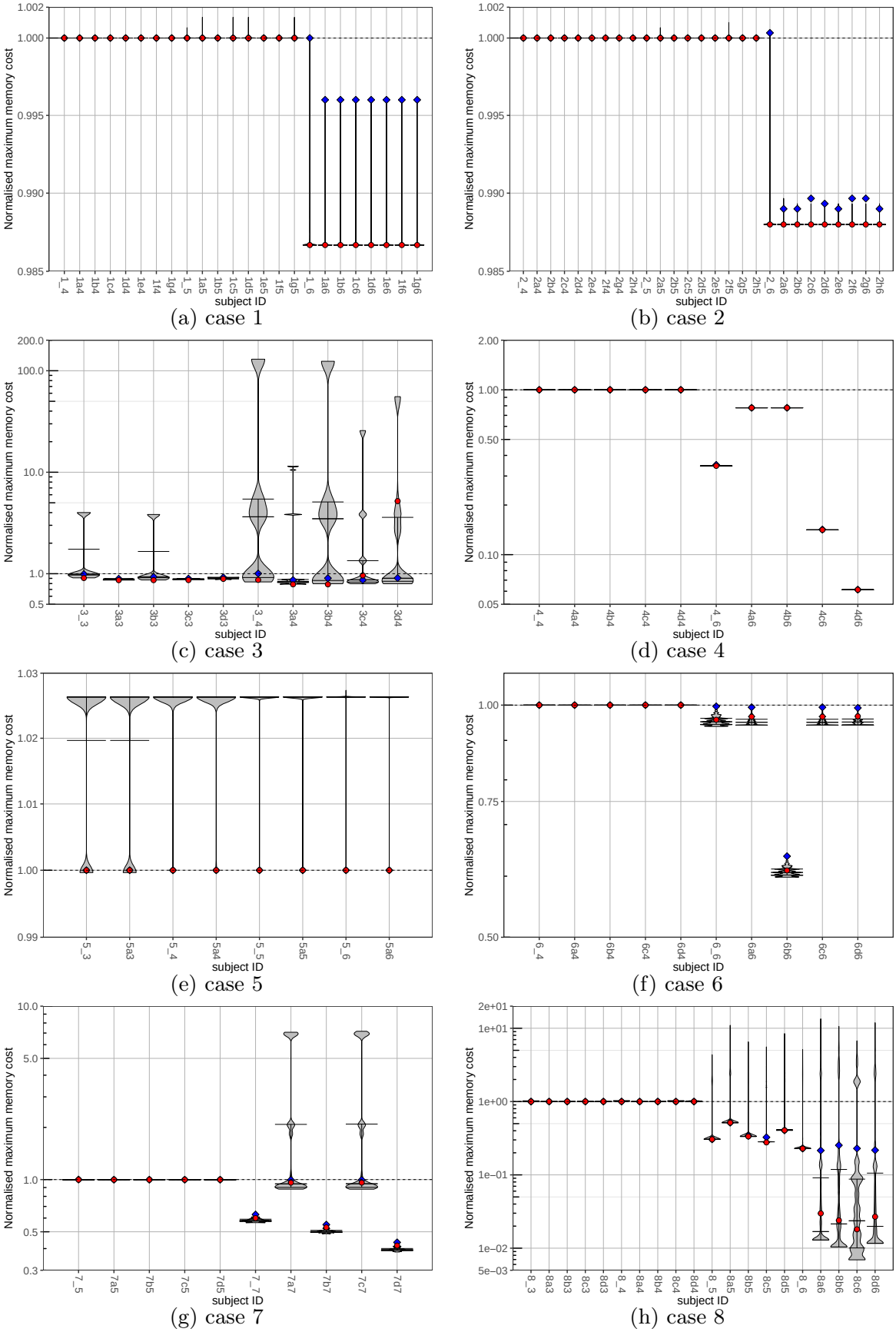
We answer this research question using Figures 5.4 and 5.5.

Aggregation order The chosen aggregation order has a major impact on the maximum memory cost and the maximum number of transitions residing in memory. Two aggregation orders may differ by several orders of magnitude depending on the subject for both normalised memory cost and normalised maximum number of transitions.

Although the maximum number of transitions generated may differ with the monolithic approach by several orders of magnitude, this does not naturally translate to normalised maximum memory cost. This is evident in the ranges that both metrics cover: in many cases the ranges over the normalised maximum number of transitions (Figure 5.5) cover several orders of magnitude while the range over the normalised maximum memory cost (Figure 5.4) is only marginal in comparison. This is likely caused by the state space compression methods that CADP employs resulting in a somewhat unpredictable relation between the two metrics.

Scaling In general we observe that the range covered by the distribution of aggregation orders increases as the number of parallel processes increases. In all cases scaling up results in a better performance of the best aggregation orders with respect to monolithic verification, i.e., as the subjects increase in size, compositional aggregation becomes increasingly viable. The only exception is the normalised memory consumption in case `5` where the range does not change with scale. For both response variables the range extends both upwards and downwards in cases `3` and `8`; compared to the smaller subjects (in scale) the bad aggregation orders become worse and the good aggregation orders better. In cases `7` and `1` this is also observed for the normalised maximum memory cost

[8] All generated data is available at `http://www.win.tue.nl/mdse/composition/test_cases_data.zip`.



(a) case 1 (b) case 2

(c) case 3 (d) case 4

(e) case 5 (f) case 6

(g) case 7 (h) case 8

Figure 5.4: Distribution of the *normalised memory cost* generated by the aggregation orders per subject (violin plots) and case (sub-figures)

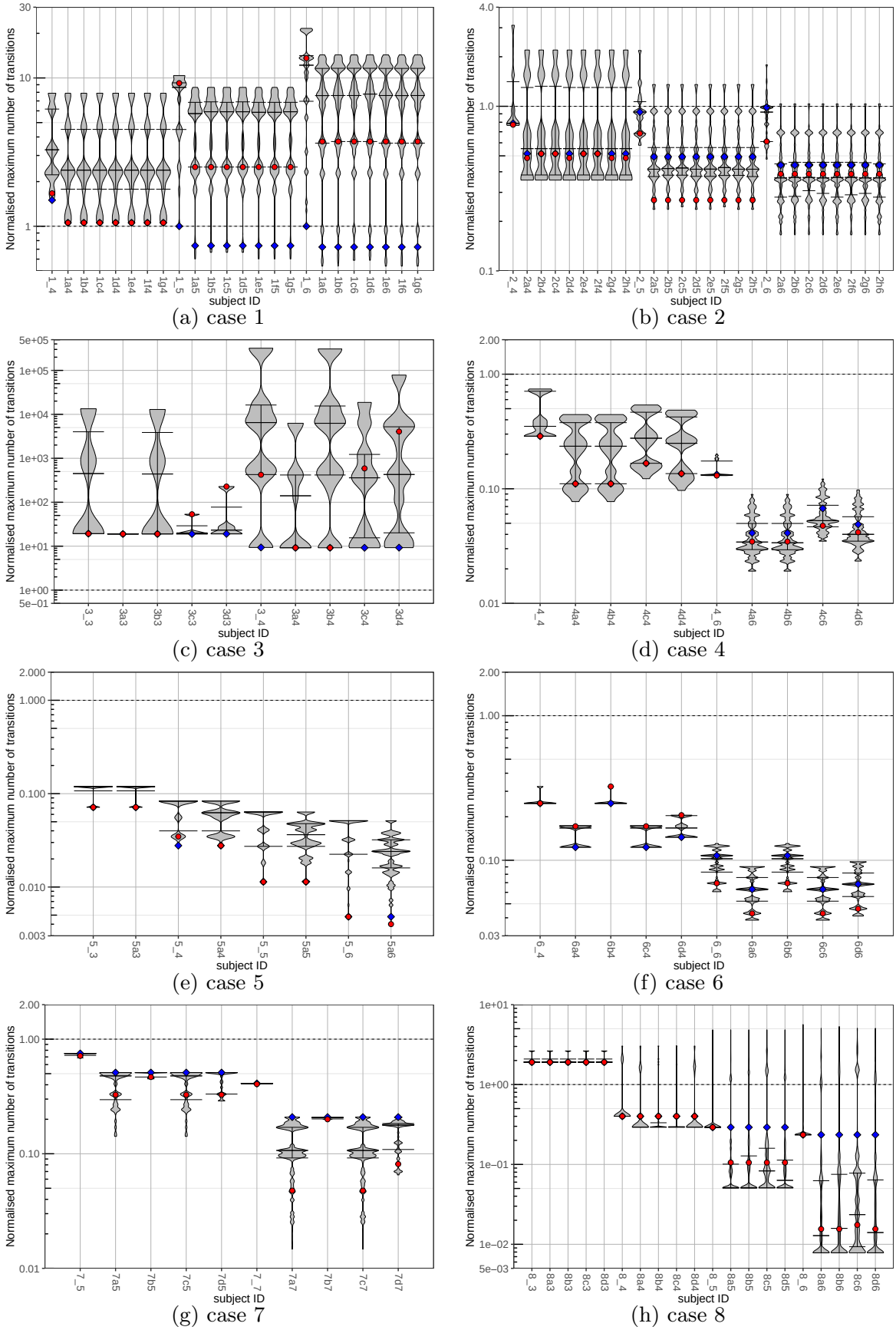


Figure 5.5: Distribution of the *normalised maximum number of transitions* generated by the aggregation orders per subject (violin plots) and case (sub-figures)

and the normalised maximum number of transitions, respectively. In the remaining cases the whole range shifts downwards as the number of parallel processes increases (again with exception of 5 for normalised maximum memory cost). Finally, in cases 1 and 2 the range of the normalised maximum memory cost increases slightly at 5 parallel processes compared to 4, and suddenly drops at 6 parallel processes. As similar results are not visible in the normalised maximum number of transitions, it is likely that the state space compression used by CADP is the cause of this.

The shape of the distributions tends to change as the number of parallel processes increases, this is most notable in the maximum number of transitions. One of the factors contributing to this phenomenon is the increasing number of data points in the distributions as the scale increases; there are 4, 26, and 236 distinct aggregation orders at 3, 4, and 5 parallel processes, respectively. At larger scales a sample of 2,500 orders was taken. This effect is particularly visible in case 3, where the model at scales 3 and 4 are compared. However, most likely the changes are due to the number of repeated processes. Due to this the balance of constituents of the model changes causing the high density areas to change accordingly.

Action hiding One would think that applying action hiding practically always results in an improvement. While this is indeed mostly true for the maximum number of transitions, this varies heavily for the maximum memory cost. The memory consumption (Figure 5.4) may remain unchanged or even increase when hiding is applied: a number of subjects show this trend in cases 4 to 8. Some of the subjects in these cases do show reduced memory consumption when hiding is applied. In cases 1 and 2 action hiding always improves the worst case normalised memory consumption.

The maximum number of transitions generated (Figure 5.5) is generally affected positively from action hiding, the only exceptions being subjects 5a3, 8a3 to 8d3. In cases 1 and 2 practically no distinction in the maximum number of transitions is observed between the applied hiding sets. Cases 3, 4, 6, 7, and 8 show moderate to significant variation in performance depending on the applied hiding set. For those subjects where the hiding sets have a noticeable impact, also the shape of the distribution is affected. For instance, subject 7c5 has a higher density around the best performance value, forming a vase shape between the minimum and the first quartile, than 7d5, which has a short tail in the same area.

5.5.2.2 RQ 5.2 How effective are the aggregation orders chosen by current heuristics?

Figures 5.4 and 5.5 show how smart reduction (indicated by a red dot) and root leaf reduction (indicated by a blue diamond) relate to other aggregation orders. Both action hiding and the scaling can have a significant effect on their performance. However, there is no clear relation between these variables and the performance, which is particularly visible for case 3.

Smart reduction has at most the memory cost of monolithic minimisation in 109 out 119 subjects, Furthermore, smart reduction generates at most the number of transitions that the monolithic approach generates in 80 out of 119 subjects. Root leaf reduction performs better or equal than monolithic minimisation in 109 and 96 out of 119 subjects in terms of memory consumptions and number of transitions generated, respectively. Furthermore, smart reduction and root leaf reduction find a best aggregation order (with respect to the sample) for, respectively, 101 and 83 out of the 119 subjects in terms of

Table 5.2: Normalised (w.r.t monolithic) maximum memory cost descriptive statistics; with "Smallest" and "Largest" indicating, respectively, the smallest and largest number of parallel processes of the 8 subjects

Size	Prop. ID	Mean			Median			# < monolithic			# < other heuristic		
		smart	root	leaf	smart	root	leaf	smart	root	leaf	smart	root	leaf
Smallest	_	0.99		1.00	1.00		1.00	2		1	2		1
Smallest	a	0.98		0.99	1.00		1.00	2		2	1		0
Largest	_	0.75		0.78	0.91		1.00	7		4	6		0
Largest	a	0.81		0.85	0.96		0.99	7		7	6		0

Table 5.3: Normalised (w.r.t monolithic) maximum transitions descriptive statistics; with "Smallest" and "Largest" indicating, respectively, the smallest and largest number of parallel processes of the 8 subjects

Size	Prop. ID	Mean			Median			# < monolithic			# < other heuristic		
		smart	root	leaf	smart	root	leaf	smart	root	leaf	smart	root	leaf
Smallest	_	3.12		3.10	0.74		0.77	5		5	3		1
Smallest	a	2.89		2.91	0.41		0.51	5		5	2		1
Largest	_	54.65		1.53	0.32		0.32	6		6	4		2
Largest	a	1.68		1.36	0.05		0.22	6		7	6		1

maximum memory cost. Likewise, smart reduction and root leaf reduction find a best aggregation order (with respect to the sample) for, respectively, 29 and 40 out of the 119 subjects in terms of maximum number of transitions generated.

Since our data is obtained from repeated measurements over eight cases, to make a fair and meaningful comparison we select cases under related conditions. We select the "smallest" and "largest" *subjects* in the number of parallel processes from the subjects considered so far. From the properties we select the only two property IDs that all cases have in common; "_" (no property) and "a" (no deadlock). The intersection of these two pairs of selections yields four sets of subjects within which a comparison is made. First a comparison between the performance of the smart reduction and root leaf reduction is made, after which their performances are compared with the performance of best aggregation orders of the sample sets of the corresponding subjects.

Smart reduction versus root reduction Tables 5.2 and 5.3 compare the normalised maximum memory cost and the normalised maximum number of transitions, respectively, of smart reduction and root leaf reduction. The first two columns indicate the selection criteria for the number of parallel processes. A comparison is made between smart reduction and root leaf reduction indicated by the *smart* and *root leaf* columns. The *Mean*, *Median* columns show the mean and median normalised maximum transitions. The final two columns, # < *monolithic* and # < *other heuristic*, indicate in how many cases the heuristics perform better than monolithic and the other heuristics, respectively.

Mean. In terms of *maximum memory consumption* there is little difference between the means of smart and root leaf reduction for all groups of subjects. For both heuristics the mean performance is slightly lower or equal to monolithic minimisation for the groups of "smallest" subjects. Both heuristics have a mean maximum memory cost between 0.75 and 0.85 times that of the monolithic approach in the groups of "largest" subjects, where hiding all actions (as implied by the no deadlock property) seems to decrease the effectiveness.

In terms of *maximum number of transitions* generated there is little difference between the means of smart reduction and root leaf reduction in the groups of "smallest" subjects. For both heuristics the mean performance is around 3 times that of the monolithic approach. In the groups of "largest" subjects there is significant difference between the means of smart reduction and root leaf reduction in group "_". In group "a" this difference is only 0.32 in favor of root leaf reduction. The high mean value for smart reduction is caused by its poor performance at cases 1 and 3.

Although the mean maximum number of transitions generated may be several orders larger, this does not seem to affect the mean maximum memory cost much. To draw any conclusions we must look back to the individual subjects in Figures 5.4 and 5.5. Indeed, we can observe that in nearly all cases both heuristics are able to find aggregation orders that perform equally well or better than the monolithic approach in terms of maximum memory cost even in subjects where the maximum number of transitions generated by the heuristics is much higher than that of the monolithic approach (see 1_6, all subjects of case 3, and all subjects of case 8 at scale 3). The converse is observed in case 5 where the maximum number of transitions generated is lower than that of the monolithic approach, but compositional aggregation does not seems to benefit from this. Both observations are likely caused by the state space compression methods that CADP employs.

Median. The median *maximum memory costs* is exactly 1 for both heuristics in the "smallest" group. The median improves slightly in the "largest" group for the smart reduction heuristic. This effect is, however, not as clear for root leaf reduction which shows insignificant improvement, and only in group "a".

The results are much more positive in terms of *maximum number of transitions* generated, as the median is much lower than the mean for both heuristics. Smart reduction has a slightly better median performance in general, but is over four times better in the "smallest" group with property ID a.

While the heuristics have a much better median performance than monolithic minimisation in terms of maximum number of transitions generated, this effect is not as evident in maximum memory cost. Again, likely the effects are dampened by state space compression methods used by CADP.

Number of cases better than monolithic minimisation. In *maximum memory cost* both heuristics perform better that the monolithic approach in 1 to 2 out of 8 cases in the "smallest" group: these were cases 3 and 7. For small subjects most of the aggregation orders are situated around 1 in Figure 5.4. It is, therefore, likely that the heuristics will find one of those aggregation orders. Results are much better in the "largest" group where smart reduction outperforms the monolithic approach in 7 out of 8 cases for both property groups. Case 5 is the only case for which smart reduction did not perform better than the monolithic approach. Root leaf reduction seems to benefit from property hiding as it outperforms monolithic minimisation in 7 out of 8 cases in the property group "a" but only in 4 out of 8 cases in property group "_". In the former case all cases but case 5 performed better than the monolithic approach, in the latter case monolithic minimisation was outperformed in cases 4, 6, 7, and 8.

For *maximum number of transitions* generated in the "smallest" group, both heuristics perform better than the monolithic approach in 5 out of 8 cases in both property ID groups. The remaining three cases being 1, 3, and 8 for both heuristics and property ID groups. Both heuristics perform better than the monolithic approach in 6 out of 8 cases in property ID group "_", while root leaf reduction performs better in one additional case in group "a". The two remaining cases being 1 and 3, excluding case 1 in group "a" for

Table 5.4: Normalised (w.r.t the best aggregation order of the sample) maximum memory cost descriptive statistics

Size	Prop. ID	Mean		Median		# best found	
		smart	root leaf	smart	root leaf	smart	root leaf
Smallest	_	1.00	1.01	1.00	1.00	8	7
Smallest	a	1.00	1.00	1.00	1.00	8	7
Largest	_	1.02	1.05	1.00	1.01	5	2
Largest	a	1.18	2.99	1.00	1.04	5	3

Table 5.5: Normalised (w.r.t the best aggregation order of the sample) maximum transitions descriptive statistics

Size	Prop. ID	Mean		Median		# best found	
		smart	root leaf	smart	root leaf	smart	root leaf
Smallest	_	1.02	1.02	1.00	1.00	5	5
Smallest	a	1.31	1.44	1.18	1.00	4	5
Largest	_	8.14	1.24	1.07	1.01	2	4
Largest	a	2.43	6.78	1.90	1.89	2	1

root leaf reduction.

Smart reduction versus root leaf reduction. Concerning *maximum memory cost* smart reduction generally performs equally or better than root leaf reduction. In the "smallest" group the two heuristics are mostly tied. Root leaf reduction only performs better than smart reduction in the "smallest" case with property ID "_". In the "largest" group smart reduction outperforms root leaf reduction in 6 out of 8 cases for both property groups.

In terms of *maximum number of transitions* generated the performance of the heuristics is more evenly distributed: they outperform each other nearly an equal number of times. Only in the "largest" group with property ID "_" root leaf reduction has a clear advantage.

Heuristics vs. the best aggregation order Tables 5.4 and 5.5 compare the maximum memory cost and the maximum number of transitions, respectively, of the smart reduction and root leaf reduction heuristics normalised with respect to the best performing of compositional aggregation in the sample. The final columns, *# best found*, indicate how many times a best aggregation order was found.

In both tables if we go from the "smallest" groups to the "largest" groups, both the means and medians increase, and the number of best orders found decreases. This may indicate that it becomes harder for the heuristics to find (near-)optimal aggregation orders as the number of parallel processes increases, however, this should be confirmed by further experiments.

In terms of *maximum memory cost* both heuristics are close to the best performance in a considerable number of cases; only in the "largest" group with property ID "a" we observe an increase in relative memory cost (in particular root leaf reduction). As the median performance is still close to the best of the sample, the increase of the mean is caused by bad performance on a few cases. Upon inspection this appeared to be case 8, which requires 2.31 and 16.65 times more memory than the best aggregation order for smart reduction and root leaf reduction, respectively. The remainder of the cases required at most 13% more memory than the best aggregation order of the case's sample.

The heuristics perform a few orders of magnitude worse than the best aggregation

order in terms of the *maximum number of transitions* generated. For the cases in the "smallest" group both smart reduction and root leaf reduction find a best aggregation order in 4 to 5 out of the 8 cases. Smart reduction strays from the best order mostly in the "largest" group when no hiding is applied (property group "_"), while for root leaf reduction this is observed in when hiding is applied (property group "a"). In property group "_" smart reduction performs badly in cases 1 and 3 at, respectively, 13.59 and 45.12 times the performance of the best order of the sample. In property group "a" smart reduction performs particularly worse than the best order or the sample in cases 1 (6.92 times the best) and 7 (3.24 times the best), while root leaf reduction performs reasonably well in case 7 (1.34 times the best). At the same time root leaf reduction performs badly at cases 7 (14.20 times the best) and 8 (30.03 times the best), while smart reduction measures 1.98 times the maximum number of transitions generated for case 8. Considering the maximum number of transitions generated within the context of the cases considered here, it seems that smart reduction and root leaf reduction may be preferable in different cases.

5.5.2.3 RQ 5.3 How can the success or failure of compositional aggregation be explained?

As we have seen CADP makes use of a compression scheme to reduce the state space. As the inner workings of this scheme have not been revealed in the literature it is hard to reason about relation between compositional aggregation and the corresponding memory cost. Hence, to answer this research question, we focus on the maximum number of generated transitions.

Although our experiment involves a large number of subjects, the number of different cases per topology is still rather limited. However, based on this data, we make the following observations, backed up by results obtained for additional models with the same topology that we constructed to focus on specific key aspects of the cases.

Two factors seem to be most influential regarding the effectiveness of compositional aggregation: the amount of internal behaviour within single process LTSs, and the amount of synchronisation among the process LTSs. In the latter case, the involvement of data has a noticeable effect, in particular the size of the data domain; for instance, when synchronisation on a Boolean value is specified, the receiver only needs to be able to synchronise on **true** and **false**, while the synchronisation on a Byte value already requires 256 transitions, many of which may be unnecessary in the complete model, since they handle values on which synchronisation actually never happens. However, if in an aggregation order, this receiver is selected before the corresponding sender, then in each step before selecting the sender, all 256 transitions of the receiver will remain, and interleave with the transitions of all LTSs that *are* added to the composition.

Among the subjects, case 3 demonstrates best that the involvement of a lot of (to be synchronised) data has a negative effect on compositional aggregation. Additional experiments with a simple pipes and filters model, one with a data domain ranging from 1 to 2 and the other from 1 to 100, underline this observation, the latter performing an order of magnitude worse than the former. Furthermore, the former performs very well compared to monolithic verification, demonstrating that the bad performance of compositional aggregation is not inherent to the pipes and filters topology.

The positive effect of involving a property to be checked, and therefore action hiding, demonstrates the importance of internal behaviour in the process LTSs, as action hiding adds internal behaviour. It seems of little importance which property is actually added,

i.e., whether it allows abstraction from all actions in the case of deadlock detection, or only a subset. This is best demonstrated by the token ring cases, i.e., cases 4 and 5. We manipulated case 5 in two different ways: increasing the amount of synchronisation, and increasing the amount of process-local (but not hidden) behaviour. The results clearly show that the former has a negative impact on performance, while the latter results in much better performance (by two orders of magnitude) iff a property is involved that allows the additional behaviour to be abstracted away, such as deadlock freedom.

The mutual exclusion algorithms, i.e., cases 6 and 7, have exactly the same set of properties. Those results demonstrate that the effect of adding a property is not always the same for all models of the same topology; adding a property seems to have a bigger effect on case 7 than case 6, resulting in a bigger range between the worst and best performing aggregation orders.

In a follow-up experiment, we will extend the number of cases and/or subjects per topology, to achieve conclusive evidence that could generalise these observations.

5.6 Results – RQ 5.4

5.6.1 Considered variables

The main response variables are classifiers indicating which minimisation strategy performed best for a given model in terms of maximum generated transitions and memory cost; these are called *best number of maximum transitions* and *best maximum memory cost*, respectively. Ties are resolved by selecting the strategy with the lowest overhead: monolithic minimisation has no overhead and root leaf reduction has some overhead, while smart reduction has the highest overhead.

Other response variables of interest are the *normalised maximum number of generated transitions* and *normalised maximum memory cost* (normalised with respect to monolithic minimisation) for both smart reduction and root leaf reduction. These variables give aim to predict how much more (or less) effective smart reduction or root leaf reduction are than the monolithic approach.

Given an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ the following metrics were considered as predictor variables: $HM(Ac(\mathcal{V}))$, $IM(Ac(\mathcal{V}))$, $|\mathcal{T}_\Pi|$, $SD(Ac(\mathcal{V}))$, and $WHSD(Ac(\mathcal{V}))$ (metrics are discussed at the end of Section 5.3). These metrics, calculated over elements of the network, produce a list of values. In order to test and apply these lists of metrics to correlation and learning, they must be aggregated to a single value; for this the *minimum*, *maximum*, *mean*, *median*, *standard deviation*, *sum*, and *product* were calculated over the lists. These aggregations characterise the distribution of the metrics in different ways.

In addition, metrics on the topology of the network were considered: *edge density*, *edge connectivity*, *mean distance between nodes*, and *weighted diameter* (weighted by HM, IM, CM, and $|\mathcal{T}_\Pi|$). These metrics all measure a form of connectedness of the network. In general compositional aggregation strategies should work better for loosely connected networks. Hence, these metrics are potential predictors for the success of the compositional aggregation heuristics.

5.6.2 Case and subject descriptions

Table 5.6 presents a summary of the experiments. The *first* column indicates the minimisation strategy. The *# finished* column indicates the number of cases that the strategy was able to minimise. The *# best max. transitions* columns show the number of

Table 5.6: Descriptive statistics of smart reduction, root leaf reduction, and monolithic minimisation on the 1,615 cases

Min. strategy	**# finished**	**# best max. transitions**		**# best max. memory cost**	
		equal priority	**prioritised**	**equal priority**	**prioritised**
Smart reduction	1,425	1,267	1,108	893	876
Root leaf reduction	1,361	239	239	443	443
Monolithic minimisation	1,146	268	268	296	296

cases for which the strategy generated to lowest maximum number of transitions. In the *equal priority* column ties are also counted, in the *priority* column ties are resolved by selecting the strategy with the lowest overhead: monolithic minimisation has no overhead and root leaf reduction has some overhead, while smart reduction has the highest overhead. The *# best max. memory* columns give the number of cases for which the strategy had the lowest maximum memory cost. Again, in the *equal priority* column ties are counted, and in the *priority* column ties are resolved as mentioned previously.

Smart reduction, root leaf reduction, and monolithic minimisation were able to finish computations for 1,425, 1,361, and 1,146 models, respectively.

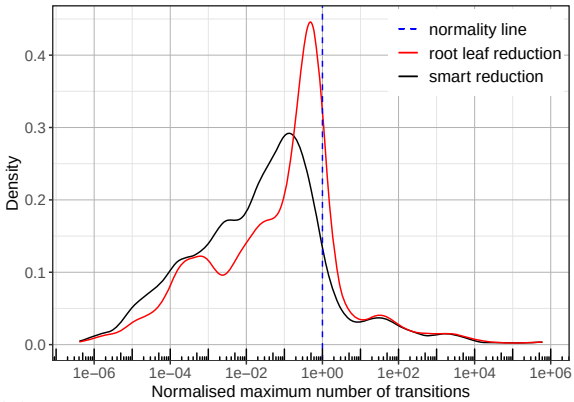
Smart reduction had a better or equal performance compared to the monolithic approach in 1,267 and 893 cases in terms of maximum number of generated transitions and maximum memory cost, respectively. In 159 and 17 cases smart reduction performed the same as monolithic minimisation or root leaf reduction for maximum number of transitions generated and maximum memory cost, respectively. Hence, taking overhead into account smart reduction performed better in 1,108 and 893 cases in terms of maximum generated transitions and maximum memory cost, respectively.

Root leaf reduction performed better than or equal to monolithic minimisation in 239 and 443 cases in terms of maximum generated transitions and maximum memory cost, respectively. In 0 of these cases root leaf reduction performed the same as the monolithic approach. Taking overhead into account this means that root leaf reduction is the best in 239 and 443 cases concerning maximum generated transitions and maximum memory cost, respectively.

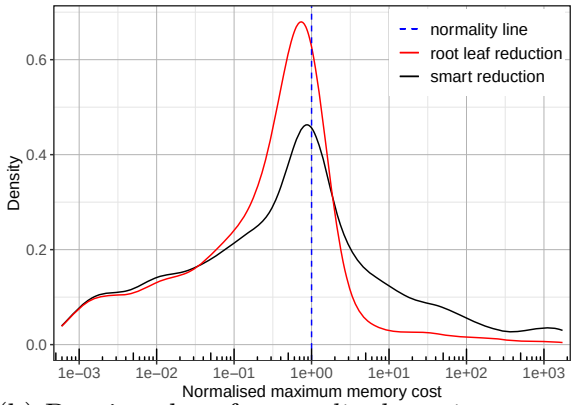
Finally, *monolithic minimisation* performed better than or equal to the two heuristics in 268 and 296 out of 1,615 cases in terms of maximum number of generated transitions and maximum memory cost, respectively. Since monolithic minimisation has the lowest computational overhead the approach is considered the best in all these cases when taking overhead into account.

As smart reduction outperforms the other approaches by a significant margin there is a class imbalance. We will take this into account when performing the data analysis.

Figure 5.6 shows the kernel density plots of *normalised maximum number of transitions* and *normalised maximum memory cost* response variables for both smart reduction and root leaf reduction. The x-axis indicates the value of the response variables and is $\log_{10}$-scale. The y-axis shows the probability that the response variable of a given case has the value indicated by the x-axis. Equal performance to monolithic minimisation, i.e., the normality line, is indicated by the *blue dashed line*. The density plots are drawn over 973 and 981 cases for smart reduction and root leaf reduction, respectively. These cases are the ones that both the corresponding heuristic and monolithic minimisation successfully completed. For other cases, the normal value with respect to the monolithic approach cannot be computed. Since the plots are drawn over the intersecting cases they may not represent the performance distribution exactly; non-overlapping data both in



(a) Density plot of *normalised maximum number of transitions*; the x-axis is $\log_{10}$-scale



(b) Density plot of *normalised maximum memory cost*; the x-axis is $\log_{10}$-scale

Figure 5.6: Density plots of *normalised maximum number of transitions* (a) and *normalised maximum memory cost* (b) for both smart reduction (*solid black line*) and root leaf reduction (*solid red line*), the *dashed blue line* indicates the point of equal performance to monolithic minimisation

favour of the heuristics and in favour of monolithic minimisation is missing. Nevertheless, they give a rough indication of the performance distributions of the heuristics.

Figure 5.6a shows that *normalised maximum number of transitions* generated by smart reduction is overall significantly more skewed to the left; this shows that smart reduction performs more often and more orders of magnitude (than monolithic minimisation) better than root leaf reduction.

Figure 5.6b shows that also in terms of *normalised maximum memory cost* the values are strongly skewed to the left; an effect made less visible by the $\log_{10}$-scale of the x-axis. It seems that root leaf reduction has a higher surface on the left of the normality line than smart reduction, however, one should note that many data points are missing that are in favour of smart reduction; a fact confirmed by the higher number of cases in which smart reduction occurs most often in *best maximum memory cost* (see Table 5.6).

5.6.3 Analysis

Analysis Procedure To perform the analysis, we first eliminate redundant variables. Then, correlation testing is applied to test the likelihood of a monotonic relation between predictor and response variables. Next, relations between response variables and the predictor variables are considered all at once through machine learning methods. Finally, the prediction models learned are validated.

Variable Elimination. Predictors that are highly correlated with other predictors introduce redundancy in the analysis. Therefore, we have selected one predictor variable among any two predictors that were highly correlated. Two predictors were considered to be highly correlated if the Kendall correlation coefficient is greater than or equal to 0.7 and the correlation is confirmed by the scatter plot of the two predictors.

Correlation Testing. As a first analysis the Kendall correlation coefficient between the *normalised maximum number of generated transitions* and *normalised maximum memory cost* response variables and the remaining predictors was calculated. The correlation

coefficient gives an indication on how strongly the relation between predictor and response variable can be captured as a monotonic relation.

Since a large range of predictor variables was tested against the response variables there is a higher chance to find ones that are correlated, this is also called the multiple comparison problem [182]. We have applied Bonferroni correction [30] to compensate for this situation.

This step could not be taken for the other two response variables as they are categorical.

Machine Learning. Machine learning methods were used to investigate whether considering multiple predictors increases the accuracy of the prediction model. Furthermore, some machine learning methods are able to learn non-linear prediction models. With this in mind we have selected a mixture of machine learning methods.

The data was partitioned into a *training set* (75% of the data) and a *test set* (the remaining 25%). Before the prediction models were trained, the data was standardised using scaling to improve model performance. *Scaling* changes the range of the data with respect to the standard deviation without affecting its distribution curve.

For regression (*normalised maximum number of generated transitions* and *normalised maximum memory cost*) the selected machine learning methods were:

- Linear methods: Linear Regression (LR), Generalized Linear Model with Elasticnet Regularization (GLMER), and Support Vector Machines with Linear Kernel (SVMwLK).
- Non-linear methods: Classification and Regression Trees (CART), K-Nearest Neighbours (KNN), Random Forests (RF), Support Vector Machines with Radial Kernel (SVMwRK), and Quantile Regression Neural Network (QRNN).

Mean Absolute Error (MAE) was chosen as the performance metric for regression methods. MAE takes the mean of the absolute value of prediction errors; indicating how concentrated the data is around the learned fit. As the response variables are normalised with respect to monolithic minimisation application of MAE to these variables is misleading: an error below 1 would weigh less than an error above 1. For instance, a prediction of 0.1 and 10 instead of 1 both differ by a factor 10, but the former would have a weight of 0.9, while the latter would have a weight of 9. Instead, we first take the $\log_{10}$ of the response variables such that errors have a more intuitive meaning, i.e., error values indicate the factor by which a prediction is off.

For classification (*best number of maximum transitions* and *best maximum memory cost*) the selected machine learning methods were:

- Linear methods: Linear Discriminant Analysis (LDA), Generalized Linear Model with Elasticnet Regularization (GLMER), and Support Vector Machines with Linear Kernel (SVMwLK).
- Non-linear methods: Classification and Regression Trees (CART), K-Nearest Neighbours (KNN), Random Forests (RF), Support Vector Machines with Radial Kernel (SVMwRK), and Learning Vector Quantization (LVQ).

As the collected data suffered from a class imbalance we opted for Cohen's κ coefficient [50] as the performance metric as opposed to accuracy. Cohen's κ coefficient measures agreement between predicted and observed values for categorical items. This metric considers the possibility of classification agreement to occur by chance. A value of 1 indicates perfect agreement, while a value of 0 indicates no agreement.

Prediction models were trained using k-fold cross validation repeated 5 times. Cross validation estimates the skill of the prediction model on unseen data. Repeated cross

validation reduces bias and variance of the prediction model.

The k-fold cross validation partitions the data in k random sets. Then for each set the prediction model is trained on the union of all models minus the given set. This withheld set is used to evaluate the model and the error estimation is remembered. Finally, the error estimations of all iterations are averaged to summarize the total effectiveness of the learning method.

Cross validation reduces bias since all of the data is used for training. Furthermore, variance is reduced since all data is also used for testing at one point. A higher number of folds reduces the partition sensitivity [120, 176]. A plateau is usually reached at around 10 folds [108, 120, 126, 176]. Furthermore, it is advisable to select a k that is a divisor of the size of the dataset such that the data is split in equally sized sets. Therefore, we have selected as k the divisor of the size of our data set that is closest to 10.

When cross validation is repeated the prediction model score is the mean of the models learned during cross validation. Repeated cross validation is less affected by the chosen partitioning as it considers a different partitioning each repetition.

The `caret` R package [125] was used to train the prediction models. The training methods of the `caret` package by default apply cross validation to tune the parameters of the machine learning methods. Repeated cross validation is applied with a number of different parameter values. After a number of repetitions the best performing iteration is chosen as the prediction model.

Validation. As validation we compare the prediction errors on the training set and the test set of the best learned models. We report the performance of the prediction models on both the repeated cross validation and the prediction errors comparison. The former gives an indication on how the learning algorithm performs on the data. The latter gives an impression of the stability with respect to unseen data and the error distribution of the final model.

For the classification models we use the models' accuracy and Cohen's κ coefficient as a first error metric. Afterwards we discuss the sensitivity, specificity, and precision of the prediction models on the test set. The *sensitivity* with respect to a class C is the probability that a case is correctly classified as C.

$$Sensitivity(C) = \frac{number\ of\ C\ elements\ predicted\ correctly}{total\ number\ of\ C\ elements}$$

The *specificity* with respect to a class C is the probability that a case is correctly classified as not being of class C.

$$Specificity(C) = \frac{number\ of\ non\text{-}C\ elements\ predicted\ correctly}{total\ number\ of\ non\text{-}C\ elements}$$

The *precision* with respect to a class C is the probability that a predicted C is correct.

$$Precision(C) = \frac{number\ of\ C\ elements\ predicted\ correctly}{total\ number\ of\ C\ elements\ predicted}$$

Variable Elimination There were 28 pairs of highly correlated predictors, all correlations were highly significant. These pairs were laid out in networks of correlated predictors and resulted in 11 disjoint networks of highly correlated predictors. Then, the most connected predictor was chosen as the representative of the network. The selection was made as follows:

1. *WHSD mean* was selected in favour of *WHSD median*, *WHSD sum*, and *WHSD product*;
2. $|\mathcal{T}_\Pi|$ *mean* was selected in favour of $|\mathcal{T}_\Pi|$ *sum*, $|\mathcal{T}_\Pi|$ *standard deviation*, $|\mathcal{T}_\Pi|$ *maximum*, and *diameter weighted by* $|\mathcal{T}_\Pi|$;
3. *mean distance between nodes* was selected in favour of *diameter weighted by HM*, *diameter weighted by IM*, and *diameter weighted by CM*;
4. *WSD median* was selected in favour of *WSD mean*;
5. *IM median* was selected in favour of *CM median*;
6. *IM mean* was selected in favour of *CM mean*;
7. *IM product* was selected in favour of *CM product*;
8. *IM minimum* was selected in favour of *CM minimum*;
9. *IM standard deviation* was selected in favour of *CM standard deviation*;
10. *HM mean* was selected in favour of *HM sum*; and
11. *HM standard deviation* was selected in favour of *HM maximum*.

We ended up with 33 predictor variables that were not highly correlated with each other.

Correlation Testing Table 5.7 presents the 10 predictors that are most strongly correlated (with p-value below 0.05) with *normalised maximum number of transitions* generated and *normalised maximum memory cost* for smart reduction and root leaf reduction. The columns indicate the rank of the predictor, the correlation coefficient of the predictor (with respect to the corresponding response variable), and the name of the predictor, respectively. Even the most strongly correlated predictors are only very weakly correlated to the corresponding response variables. This indicates a very weak monotonic relation between the predictors and response variables.

The *minimum*, *standard deviation*, and *product* of IM are among predictors that are most strongly correlated to *normalised maximum number of transitions* for both smart reduction and root leaf reduction. Other predictors in the top 10 include variants of $|\mathcal{T}_\Pi|$, CM, and $HSDV$.

For both smart reduction and root leaf reduction, *normalised maximum memory cost* is most strongly correlated to the *mean*, *product*, and *weighted diameter* corresponding to $|\mathcal{T}_\Pi|$, and the *minimum* and *standard deviation* of IM. Other predictors in the top 10 include variants of IM, CM, and $HSDV$.

To get a better impression of the relation between the predictors and the response variables we have investigated the corresponding scatter plots. The scatter plots of the strongest correlated predictor versus the corresponding response variable are shown in Figure 5.7. The response variables and predictors are laid out on the *y-axis* and *x-axis*, respectively. The y-axis has a $\log_{10}$-scale for both predictors, the x-axis for *normalised maximum number of transitions* has a continuous scale, and the x-axis for *normalised memory cost* has a continuous $\log_{10}$-scale. The normality line, i.e., the line indicating equal performance with respect to monolithic minimisation, is indicated by the *dashed blue line*. The *solid red line* indicates an estimated trend line in the data computed using LOcally WEighted Scatterplot Smoothing (LOWESS) [48]. The *grey area* around the

Table 5.7: The 10 best Kendall correlation coefficients of the 33 predictors with respect to the response variables *normalised maximum number of transitions* (a,c) and *normalised maximum memory cost* (b,d) for smart reduction (a,b) and root leaf reduction (c,d)

(a) Smart reduction – Correlation w.r.t *normalised maximum number of transitions*

#	**Coefficient**	**Predictor**
1	-0.212	*IM minimum*
2	-0.154	*IM standard deviation*
3	0.147	*IM product*
4	-0.142	$\lvert\mathcal{T}_\Pi\rvert$ *mean*
5	-0.141	*Diameter weighted by* $\lvert\mathcal{T}_\Pi\rvert$
6	-0.122	*HSDV median*
7	-0.117	*CM maximum*
8	-0.109	*Diameter weighted by HSDV*
9	-0.097	*HSDV product*
10	0.093	*WHSD mean*

(b) Smart reduction – Correlation w.r.t *normalised maximum memory cost*

#	**Coefficient**	**Predictor**
1	-0.212	$\lvert\mathcal{T}_\Pi\rvert$ *mean*
2	-0.209	*Diameter weighted by* $\lvert\mathcal{T}_\Pi\rvert$
3	0.201	*IM minimum*
4	-0.191	$\lvert\mathcal{T}_\Pi\rvert$ *product*
5	-0.158	*IM standard deviation*
5	-0.137	*HSDV median*
7	-0.131	*Diameter weighted by HSDV*
8	0.124	*IM product*
9	-0.121	*HSDV product*
10	-0.121	*CM max*

(c) Root leaf reduction – Correlation w.r.t *normalised maximum number of transitions*

#	**Coefficient**	**Predictor**
1	0.217	*IM minimum*
2	-0.187	*IM standard deviation*
3	-0.141	*CM maximum*
4	0.133	*IM product*
5	-0.124	$\lvert\mathcal{T}_\Pi\rvert$ *mean*
6	-0.124	*Diameter weighted by* $\lvert\mathcal{T}_\Pi\rvert$
7	-0.106	*IM maximum*
8	0.092	$\lvert\mathcal{T}_\Pi\rvert$ *minimum*
9	-0.085	*HSDV median*
10	-0.079	*CM sum*

(d) Root leaf reduction – Correlation w.r.t *normalised maximum memory cost*

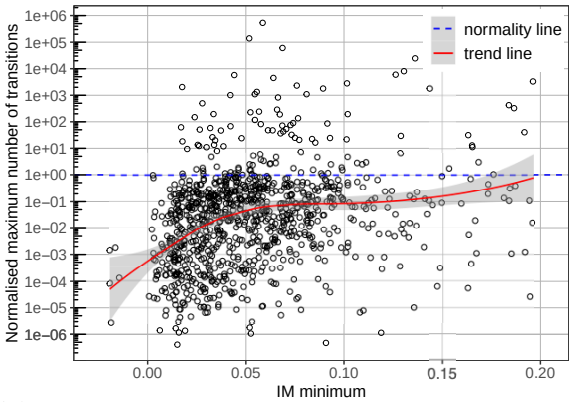
#	**Coefficient**	**Predictor**
1	-0.263	$\lvert\mathcal{T}_\Pi\rvert$ *mean*
2	-0.260	*Diameter weighted by* $\lvert\mathcal{T}_\Pi\rvert$
3	-0.225	$\lvert\mathcal{T}_\Pi\rvert$ *product*
4	0.216	*IM minimum*
5	-0.152	*IM standard deviation*
5	0.145	*IM product*
7	0.119	*IM median*
8	-0.109	*CM maximum*
9	-0.099	*Diameter weighted by HSDV*
10	-0.086	*HSDV median*

trend line indicates their 95%-confidence interval. The scatter plots against the response variables are similarly distributed for most predictors.

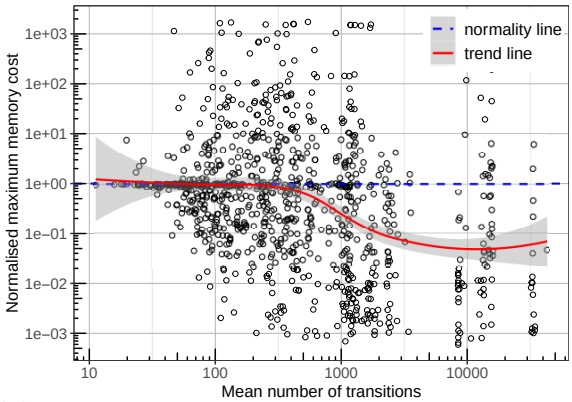
Normalised Maximum Number of Transitions. The scatter plots for both smart reduction (Figure 5.7a) and root leaf reduction (Figure 5.7c) show an upward trend: a higher value of *IM minimum* is roughly associated with a higher *normalised maximum number of transitions*. This trend is expected as high IM values indicate that an LTS network is expected to have relatively many interleaving transitions.

The confidence interval is quite narrow along most of the x-axis: wider at the extremes where relatively little data is located and especially thin between 0 and 0.1. If an IM *minimum* is between 0 and 0.1, then it is highly likely that the *normalised maximum number of transitions* generated by smart reduction and root leaf reduction will be between 1 to 1×10^4 times smaller than that of monolithic minimisation. Still, even in this range, there is an off chance that smart reduction and root leaf reduction generate 1×10^6 times the number of transitions of the monolithic approach.

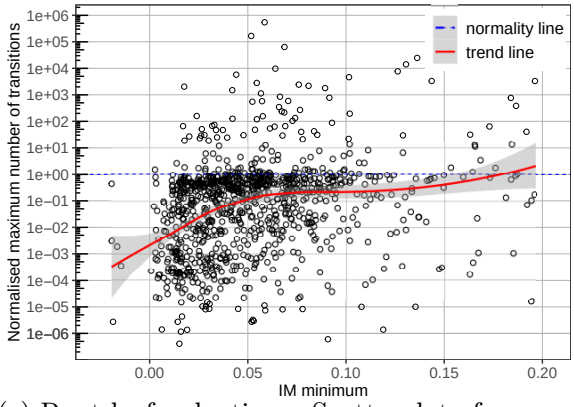
The scatter plots for both heuristics are most dense below a *normalised maximum number of transitions* value of 1. In Table 5.6 we already observed that smart reduction performs better than monolithic minimisation in most of the cases, the scatter plot of root leaf reduction shows that also root leaf reduction performs better than monolithic minimisation in most cases. Finally, for root leaf reduction most of the data is located on



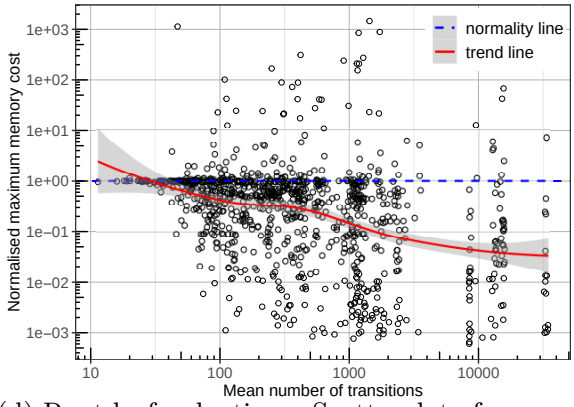
(a) Smart reduction – Scatterplot of *normalised maximum number of transitions* versus *IM minimum*; the y-axis is $\log_{10}$-scale



(b) Smart reduction – Scatterplot of *normalised maximum memory cost* versus $|\mathcal{T}_\Pi|$ *mean*; both axis are $\log_{10}$-scale



(c) Root leaf reduction – Scatterplot of *normalised maximum number of transitions* versus *IM minimum*; the y-axis is $\log_{10}$-scale



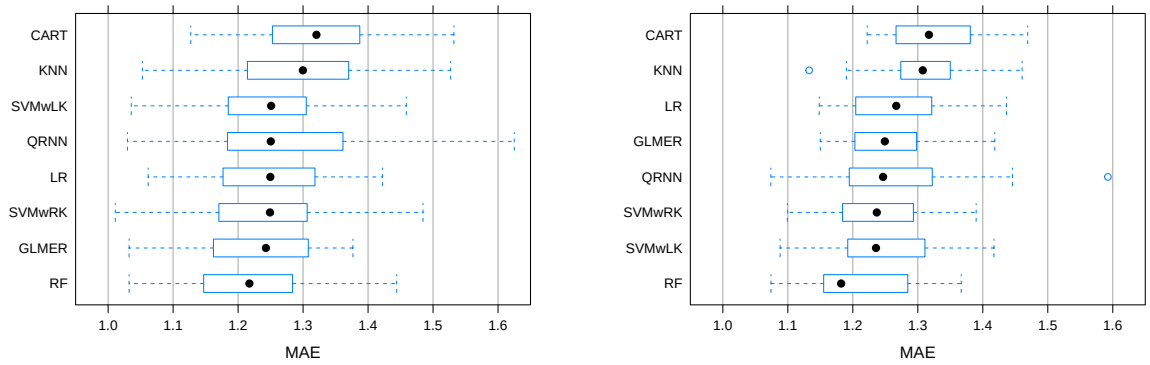
(d) Root leaf reduction – Scatterplot of *normalised maximum memory cost* versus $|\mathcal{T}_\Pi|$ *mean*; both axis are $\log_{10}$-scale

Figure 5.7: Scatter plots showing the data distribution of the predictor that is most strongly correlated to *normalised maximum number of transitions* (a,c) and *normalised maximum memory cost* (b,d) for smart reduction (a,b) and root leaf reduction (c,d); the *dashed blue line* at value 1 indicates the line of equivalent performance to monolithic minimisation, the *sold red line* indicate the estimated trend line, the *grey area* around these line indicate their 95%-confidence interval

or just below the normality line, while for smart reduction the data is spread more evenly below the normality line.

Normalised Maximum Memory Cost. For both smart reduction (Figure 5.7b) and root leaf reduction (Figure 5.7d) the scatter plots indicate a downward trend: a higher mean $|\mathcal{T}_\Pi|$ is roughly associated with a lower *normalised maximum memory cost*. We suspect that the heuristics have an advantage at higher mean $|\mathcal{T}_\Pi|$ values since they perform minimisation steps on the individual processes potentially lowering the number of transitions residing in memory.

Again, the confidence interval is quite thin along most of the x-axis. If the *mean* $|\mathcal{T}_\Pi|$ is between 30 and 10,000 for root leaf reduction and between 300 and 10,000 for smart reduction, then it is highly likely that the *normalised maximum memory cost* is between 1 to 50 times lower than that of monolithic minimisation. Smart reduction seems to perform particularly bad with respect to root leaf reduction below *mean* $|\mathcal{T}_\Pi|$ values of a



(a) Smart reduction (b) Root leaf reduction

Figure 5.8: Box plots of the performance of cross validation on *normalised maximum number of transitions* for smart reduction (a) and root leaf reduction (b) ordered by mean MAE (lower is better), with MAE in orders of magnitude

1,000. Despite the narrow confidence interval indicating a high chance to that cases are more efficient than monolithic minimisation, for both heuristics there are cases in which they require up to a 1,000 times the maximum that monolithic minimisation uses.

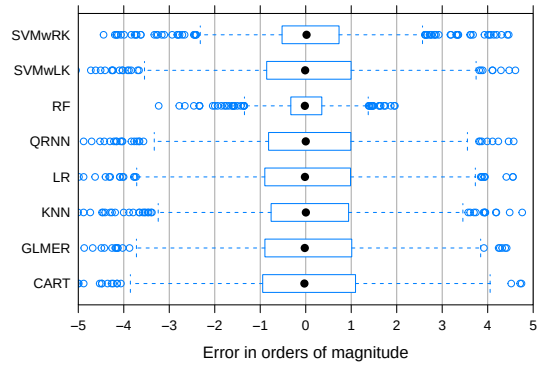
Also for the *normalised maximum memory cost* most of the data points are located near the normality line for root leaf reduction. The data points for smart reduction are spread out much more evenly. Finally, there is a lack of data in certain ranges of *mean* $|\mathcal{T}_\Pi|$; in particular between a *mean* $|\mathcal{T}_\Pi|$ of 3,000 and 7,000 there are nearly no data points. Apparently, the LTS network generator has a low chance of generating a network with a *mean* $|\mathcal{T}_\Pi|$ in this range by sampling distributions from the 88 source process LTSs. This indicates that such networks are not common in the set of generated networks. Increasing the size of the set of source process LTSs may improve variety of *mean* $|\mathcal{T}_\Pi|$.

Machine Learning & Validation

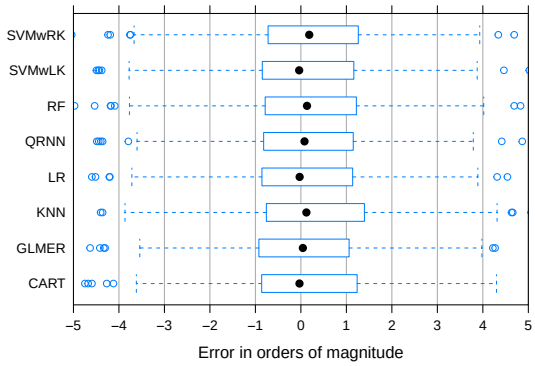
Normalised Maximum Number of Transitions. Figure 5.8 shows the performance of the cross validations in learning the *normalised maximum number of transitions* for the previously selected machine learning methods. The minimum and maximum are indicated by vertical dashed lines (these are called the whiskers). The first and third quartiles are indicated by the ends of the solid box. The median of the distribution is stipulated by the dot in the box. The box plots show the distribution of MAE in order of magnitude for each of the machine learning methods. The methods are shown in decreasing order of mean MAE. The prediction performance for *smart reduction* and *root leaf reduction* is shown in Figures 5.8a and 5.8b, respectively.

The prediction performance for *smart reduction* ranges between 1.01 (minimum for SVMwRK) and 1.62 (maximum for QRNN), these extremes indicate an MAE of factor 10^1 (≈ 10) and $10^{1.6}$ (≈ 40), respectively. GLMER and LR are the most stable as they have the least variation between the different cross validations. In terms of smallest minimum MAE the SVMwRK method is the clear winner.

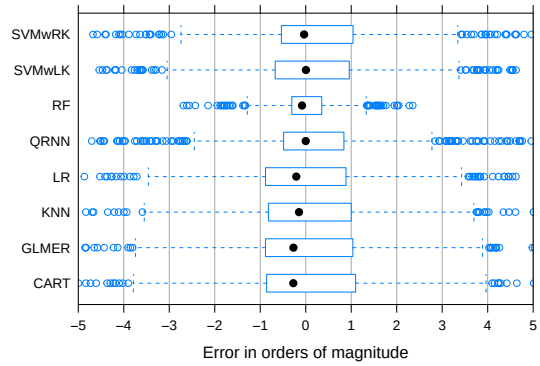
MAE for *root leaf reduction* ranges between 1.07 (minimum for RF and QRNN) and 1.59 (maximum for QRNN), these extremes indicate an MAE of approximately factor 12 and 39, respectively. There is one outlier at 105.89 for SVMwLK not shown in the Figure 5.8b. CART and GLMER are the most stable, while RF and QRNN have the



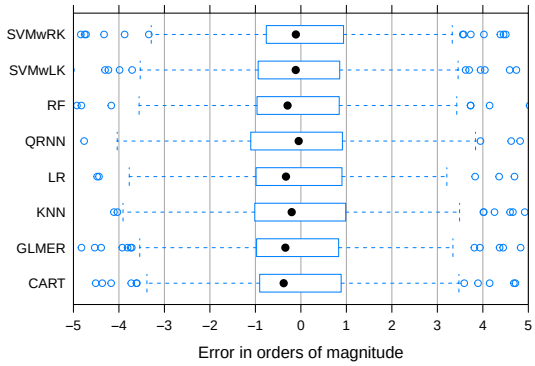
(a) Prediction errors on the *training set* for smart reduction



(b) Prediction errors on the *test set* for smart reduction



(c) Prediction errors on the *training set* for root leaf reduction



(d) Prediction errors on the *test set* for root leaf reduction

Figure 5.9: Box plots of the *normalised maximum number of transitions* prediction errors of the best learned models on the training set (a,b) and the test set (c,d) for smart reduction (a,c) and root leaf reduction (b,d) in alphabetical order

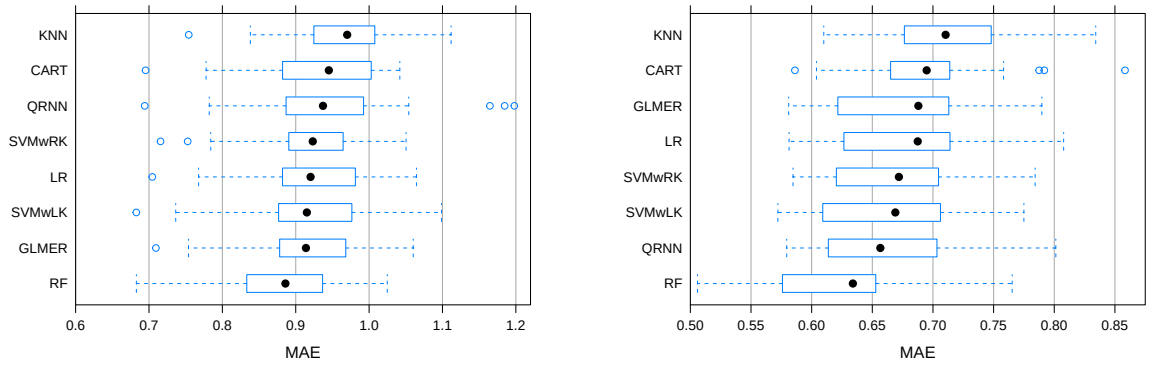
smallest minimum MAE.

Prediction of the *normalised maximum number of transitions* seems to be feasible at around one order of magnitude for both heuristics.

Next, presented in Figure 5.9 are the error distributions of the best learned prediction models on both the training set and the test set for both smart reduction (Figures 5.9a and 5.9b) and root leaf reduction (Figures 5.9c and 5.9d). Errors are displayed in orders of magnitude, where negative numbers indicate an underestimate and positive numbers an overestimate. Outliers are indicated by a circle.

For *smart reduction* the median error is about 0 in both the training and test sets. This indicates that the predictor underestimated and overestimated the values in roughly an equal number of cases. In most cases the box plots do not differ much between training set and test set, with the exception of RF. Hence, all models, with the exception of RF, tend to perform as expected on unseen cases.

The first and third quartiles of all prediction models are between -1 and 1.4; this indicates that in half of the cases the prediction is off by no more than a factor 25. Based on the test set, the best prediction models are QRNN, LR, and GLMER; these have a minimum error of -4.49, -4.63, and -4.59, respectively, a first quartile at -0.82, -0.92, and -0.86, respectively, a third quartile at 1.15, 1.06, and 1.14, respectively, and a maximum



(a) Smart reduction (b) Root leaf reduction

Figure 5.10: Box plots of the performance of the cross validations on *normalised maximum memory cost* for smart reduction (a) and root leaf reduction (b) ordered by mean MAE (lower is better), with MAE in orders of magnitude

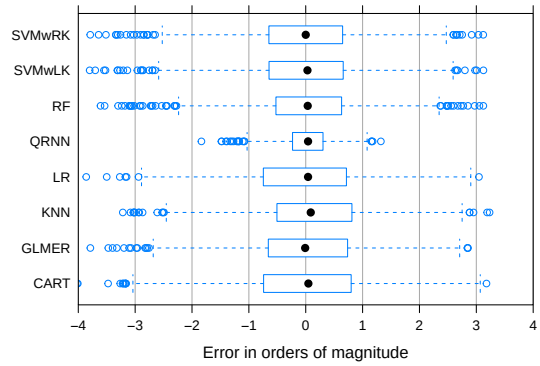
error of 5.90, 5.81, and 5.91, respectively (error in orders of magnitude). For these three prediction models, in half of the cases the models underestimate no more than a factor 9, and overestimate no more than a factor 15. Disregarding outliers QRNN, LR, and GLMER have a minimum of -3.60, -3.72, and -3.54, respectively and a maximum of 3.79, 3.89, and 3.98, respectively. When disregarding outliers LR seems to perform worse than QRNN and GLMER. Finally, the range of QRNN is slightly smaller than that of GLMER. Hence, QRNN should be preferred over GLMER.

The median error for *root leaf reduction* ranges between -0.27 and 0 in both the training and test sets with more prediction models having a slightly negative median. Most prediction models underestimate slightly more cases than they overestimate. Between training set and test set the box plots for RF and QRNN differ significantly. The remainder of the machine learning methods seem stable with respect to unseen cases.

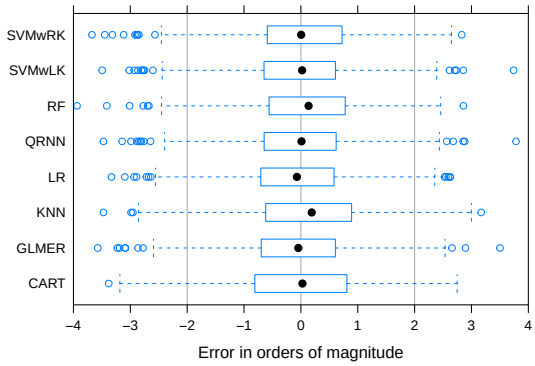
The first and third quartiles of all prediction models are between -1.1 and 1; this indicates that in half of the cases the prediction is off by no more than a factor 13. Based on the test set, the best prediction models are SVMwRK and CART; these have a minimum error of -6.16 and -6.58, respectively, a first quartile at -0.76 and -0.90, respectively,, a third quartile at 0.93 and 0.88, respectively, and a maximum error of 4.50, and 4.72, respectively (error in orders of magnitude). The minimum and maximum values of SVMwRK and CART are outliers not shown in Figure 5.9d. For these two prediction models, in half of the cases the models underestimate no more than a factor 8, and overestimate no more than a factor 9. Removing outliers, SVMwRK and CART have a minimum of -3.29 and -3.38, respectively, and a maximum of 3.33 and 3.47, respectively. When disregarding outliers SVMwRK seems to outperform CART as it has a smaller range.

The five *most important variables* (and their weight) of the best model for smart reduction (QRNN) are: *IM minimum* (100), *IM standard deviation* (62), *IM median* (41), *IM mean* (36), and *diameter weighted by HSDV* (29). The five most important variables (and their weight) of the best model for root leaf reduction (SVMwRK) are: *IM minimum* (100), *IM standard deviation* (76), *IM median* (53), *CM maximum* (31), and *IM mean* (27).

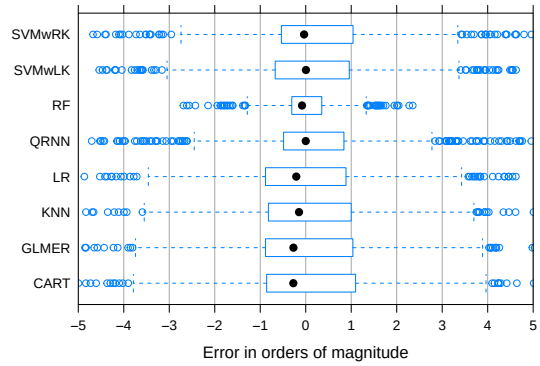
Normalised Maximum Memory Cost. Figure 5.10 shows the performance of the cross



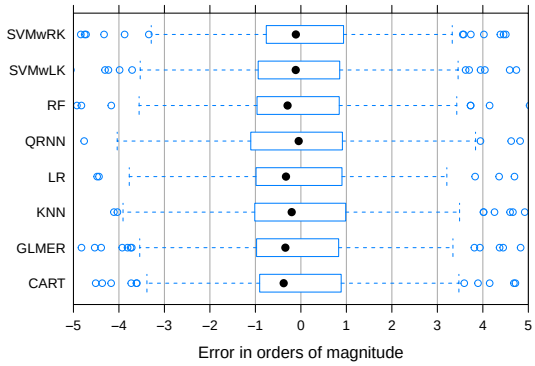
(a) Prediction errors on the *training set* for smart reduction



(b) Prediction errors on the *test set* for smart reduction



(c) Prediction errors on the *training set* for root leaf reduction



(d) Prediction errors on the *test set* for root leaf reduction

Figure 5.11: Box plots of the *normalised maximum memory cost* prediction errors of the best learned models on the training set (a,b) and the test set (c,d) for smart reduction (a,c) and root leaf reduction (b,d) in alphabetical order

validations in learning the *normalised maximum memory cost* for the previously selected machine learning methods. The box plot shows the distribution of MAE in order of magnitude for each of the machine learning methods. The methods are shown in decreasing order of mean MAE. The prediction performance for *smart reduction* and *root leaf reduction* is shown in Figures 5.10a and 5.10b, respectively.

The prediction performance for *smart reduction* ranges between 0.68 (minimum for SVNwLK and RF) and 1.20 (maximum for QRNN), these extremes indicates an MAE of factor $10^{0.68}$ (≈ 5) and $10^{1.20}$ (≈ 16), respectively. CART and WVMwRK are the most stable as they have the least variation between the different cross validations. Considering the smallest minimum MAE, the method RF performs the best.

MAE for *root leaf reduction* ranges between 0.51 (minimum for RF) and 0.86 (maximum for CART), these extremes indicate an MAE of approximately a factor 3 and 7, respectively. CART is the most stable, while RF has by far the smallest minimum MAE.

Prediction of the *normalised maximum number of transitions* seems to be feasible with an MAE of less than one order of magnitude.

Next, shown in Figure 5.11 are the error distributions of the best learned prediction models on both the training set and the test set for both smart reduction (Figures 5.11a and 5.11b) and root leaf reduction (Figures 5.11c and 5.11d). Again, errors

are displayed in orders of magnitude, where negative numbers indicate an underestimate and positive numbers an overestimate.

Similar to the *maximum number of transitions*, the median error with respect to prediction of *maximum memory cost* for *smart reduction* is about 0 in both the training and test sets, indicating that the prediction model underestimated and overestimated the values in roughly an equal number of cases. In most cases the box plots do not differ much between training set and test set, with the exception of SVMwRK and RF. Thus, with exception of these prediction models, all models tend to perform as expected on unseen cases.

The first and third quartiles of all prediction models are between -0.75 and 0.81; this indicates that in half of the cases the prediction is off by no more than a factor 7. Based on the test set, the best prediction models are SVMwLK and QRNN; these have a minimum error of -3.49 and -3.47, respectively, a first quartile at -0.63 and -0.64, respectively, a third quartile at 0.61 and 0.62, respectively, and a maximum error of 3.74 and 3.78, respectively (error in orders of magnitude). For these three prediction models, in half of the cases the models underestimate and overestimate by no more than a factor 5. The minimum and maximum values of SVMwLK and QRNN, removing outliers they have a minimum of -2.44 and -2.40, respectively, and a maximum of 2.39 and 2.44, respectively. There is no strong distinction between these two prediction models.

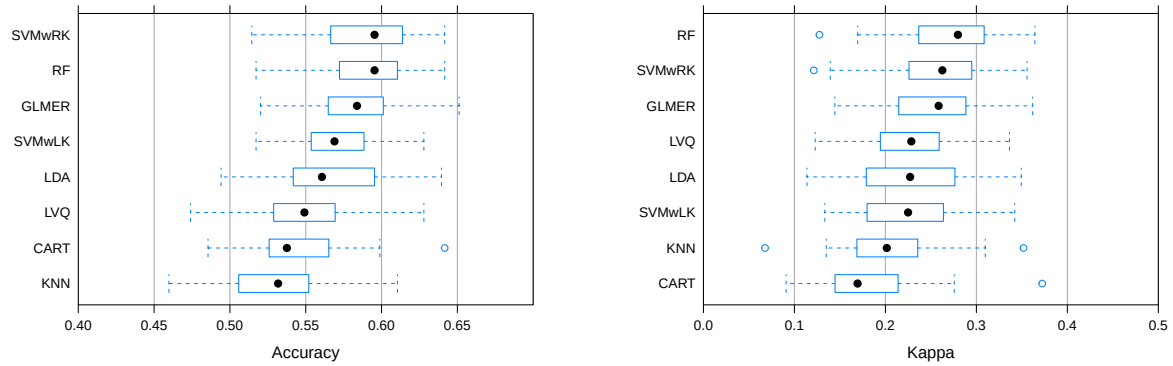
The median error for *root leaf reduction* ranges between -0.27 and 0 in the training set and between -0.37 and 0 in the test set. Hence, the models tend to underestimate more cases than they overestimate. Between training set and test set the box plots for RF and QRNN differ significantly. The remainder of the machine learning methods seem stable with respect to unseen cases.

The first and third quartiles of all prediction models are between -0.49 and 0.51; this indicates that in half of the cases the prediction is off by no more than a factor 4. Based on the test set, the best prediction models are SVMwRK, GLMER, and CART; these have a minimum error of -3.59, -2.96 and -3.93, respectively, a first quartile at -0.39, -0.62 and -0.42, respectively, a third quartile at 0.55, 0.43 and 0.32, respectively, and a maximum error of 2.67, 3.02, and 2.52, respectively (error in orders of magnitude). For these three prediction models, in half of the cases they underestimate and overestimate by no more than a factor 5. Removing outliers SVMwRK, GLMER, and CART have a minimum of -1.81, -1.89 and -1.50, respectively, and a maximum of 1.90, 1.95 and 1.42, respectively. CART performs particularly well when outliers are not considered, the chance that its predictions are off by more than a factor 31 are negligible.

The five *most important variables* (and their weight) of the best model for smart reduction (QRNN) are: $|\mathcal{T}_\Pi|$ *mean* (100), *diameter weighted by* $|\mathcal{T}_\Pi|$ (90), *IM minimum* (84), *IM standard deviation* (70), and *HSDV median* (46). The five most important variables (and their weight) of both the best model for root leaf reduction (CART) are: *IM standard deviation* (100), *IM minimum* (94), $|\mathcal{T}_\Pi|$ *product* (93), $|\mathcal{T}_\Pi|$ *mean* (72), and *diameter weighted by* $|\mathcal{T}_\Pi|$ (71).

Best Maximum Number of Transitions. Prediction models for the *best maximum number of transitions* response variable aim to predict which minimisation method is most effective for a given LTS network in terms of maximum number of generated transitions. Figure 5.12 shows the performance of the cross validations in learning the *best maximum number of transitions*. The box plots show the distribution of accuracy (Figure 5.12a) and Cohen's κ coefficient (Figure 5.12b).

The *no information rate* of the training data is 0.67; this indicates that any prediction



(a) Accuracy of the cross validations (b) Cohen's κ coefficient of the cross validations

Figure 5.12: Box plots of the accuracy (a) and Cohen's κ coefficient (b) of the cross validations on *best maximum number of transitions* ordered by the corresponding performance metric (higher is better)

Table 5.8: Performance of the best learned prediction models with respect to *best maximum number of transitions* for each prediction method on the *test set*; the classes "smart", "rt.leaf", and "mono." stand for *smart reduction*, *root leaf reduction*, and *monolithic minimisation*, respectively

Model	**Acc.**	κ	**Sensitivity**			**Specificity**			**Precision**		
			smart	**rt.leaf**	**mono.**	**smart**	**rt.leaf**	**mono.**	**smart**	**rt.leaf**	**mono.**
SVMwRK	**0.61**	**0.22**	0.85	0.24	0.30	0.39	0.87	**0.94**	0.67	**0.40**	**0.47**
SVMwLK	**0.74**	0.17	0.98	0.00	0.22	0.18	**1.00**	0.96	0.76	-	**0.44**
RF	0.73	**0.20**	0.94	0.05	0.28	0.26	0.99	0.93	0.77	0.43	0.38
LVQ	0.68	0.10	0.89	0.07	0.17	0.22	0.95	0.93	0.75	0.19	0.26
LDA	0.72	0.14	0.95	0.09	0.15	0.17	0.98	0.96	0.75	0.42	0.36
KNN	0.69	0.18	0.87	**0.16**	0.24	**0.32**	0.95	0.90	0.77	0.33	0.28
GLMER	0.73	0.19	0.95	0.07	0.24	0.23	0.99	0.94	0.76	**0.57**	0.39
CART	0.72	**0.20**	0.92	0.00	**0.33**	0.31	**1.00**	0.89	**0.78**	-	0.32

model that has an accuracy above 67% performs better than a random guess based on the distribution of the data.

The SVMwRK and RF methods have the highest median accuracy (0.69) and RF has the highest median κ (0.21). SVMwRK and LVQ have the highest maximum accuracy (0.70) and κ (0.37), respectively. The best prediction models among the cross validations have an accuracy between 0.69 and 0.74, and a κ between 0.16 and 0.37. Considering the no information rate, all of these best prediction models have a moderate accuracy and a weak inter-rater agreement.

Next, we investigate the performance of the best prediction models amongst the cross validations with respect to the training and test sets. Table 5.8 shows, for each prediction model, its *accuracy* (Acc. column) and κ coefficient, and its sensitivity, specificity, and precision with respect to the classes "smart" (for smart reduction), "rt.leaf" (for root leaf reduction), and "mono." (for monolithic minimisation). The best scores are indicated in bold.

The accuracy and κ on the training set are, respectively, 0.72 and 0.24 for SVMwRK, 0.71 and 0.23 for SVMwLK, 1.00 and 1.00 for RF, 0.70 and 0.26 for LVQ, 0.70 and 0.22 for LDA, 0.75 and 0.43 for KNN, 0.70 and 0.24 for GLMER, and 0.71 and 0.26 for CART (in the same order as the models appear in Table 5.8). RF and KNN are the least stable

in their accuracy; differencing more than 0.05 from training set to test set. The other models are relatively stable with an accuracy difference no more than 0.05. In all models, except for SVMwLK and GLMER, the κ is significantly lower in the test set than in the training set. The least amount of change is seen for the models SVMwRK and GLMER where the κ is lowered from 0.23 to 0.17 and from 0.24 to 0.19, respectively. Despite that RF seems to have over fitted on the training set it still has a reasonably high performance on the unseen test set.

We now focus the discussion on the performance of the prediction models on the test set (see Table 5.8). The SVMwRK, SVMwLK, RF, KNN, GLMER, and CART prediction models perform best on at least one aspect on the unseen test set (see bold numbers in Table 5.8).

Very high *sensitivity* for the "smart" class is achieved by all prediction models, the lowest sensitivity is 0.87 and the highest is 0.99. SVMwRK is the most sensitive to the "smart" class: it correctly classifies 99% of the cases that perform best with smart reduction. The "rt.leaf" and "mono." classes are predicted correctly at a very low to low probability with sensitivity ranging from, respectively, 0.00 to 0.16, and 0.15 to 0.32. KNN has the highest probability of predicting the "rt.leaf" correctly, namely a probability of 0.16. With a probability of 0.33 CART has the highest probability to predict the "mono." class correctly. Less than 33% of the training class consists of cases that belong to "rt.leaf" or "mono.", the low sensitivity to these classes may be caused by a lack of data.

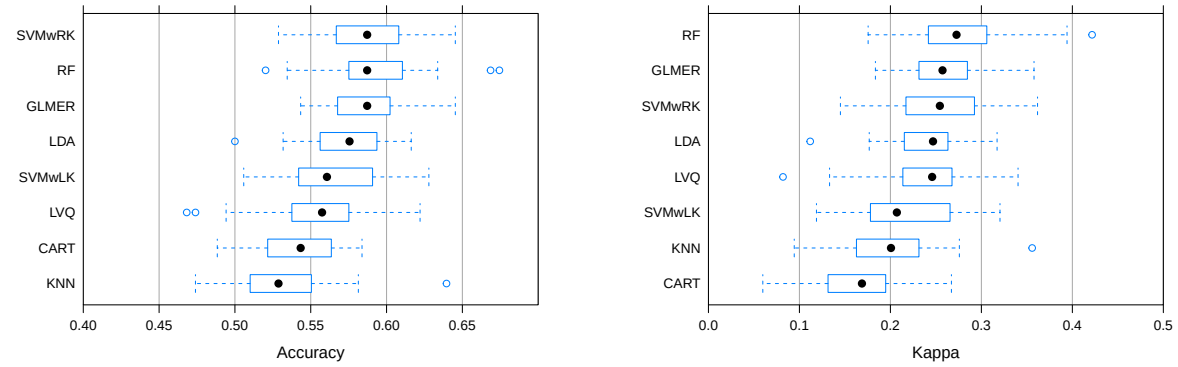
Specificity for the "smart" class is low: the probability of correctly predicting that a case should not be classified as "smart" ranges from 0.16 to 0.32. KNN has the highest specificity to the "smart" class at 32%. Specificity is significantly better for the "rt.leaf" and the "mono." classes where specificity ranges from 0.95 to 1.00 for the root leaf reduction and from 0.90 to 0.97 for monolithic minimisation. Specificity for "rt.leaf" is the highest for SVMwRK, SVMwLK, and CART, while it is highest for "mono." for SVMwRK. The specificity for the "smart" class is the worst and the specificity for the "rt.leaf" class is the best. Most of the cases in the training set belong to the "smart" class and the fewest of the cases belong to the "rt.leaf" class. Hence, there are relatively few cases *not* belonging to the "smart" class. It is, therefore, likely that results can be improved by adding more cases to the training set that are not of the "smart" class.

The prediction models have a moderate to high *precision*. The probability that a case predicted as "smart" is indeed a case belonging to the "smart" class ranges from 0.75 to 0.78. The highest probability (0.78) is achieved by CART. The precision for the "rt.leaf" class ranges from 0.19 to 0.57, where GLMER has the highest precision. The precision for the "mono." class ranges from 0.26 to 0.44 with SVMwLK achieving the highest precision.

With the current data it seems feasible to predict cases that will perform best with smart reduction. The sensitivity, and precision for the other classes is too low. There are, however, indications that more data may improve sensitivity and precision for root leaf reduction and monolithic minimisation.

The five *most important* variables for the "smart" and "rt.leaf" classes were: *HM mean*, *IM product*, *IM mean*, *IM min*, and *IM median* (in descending order of importance). For the "mono." class the five most important variables were: *HM mean*, *HM median*, *IM median*, $|\mathcal{T}_\Pi|$ *mean*, and *IM median* (in descending order of importance).

Best Maximum Memory Cost. Prediction models for the *best maximum memory cost* response variable aim to predict the most effective minimisation method for a given LTS network in terms of maximum memory cost. Figure 5.13 shows the performance of the



(a) Box plots of the accuracy of the cross validations

(b) Box plots of the Cohen's κ coefficient of the cross validations

Figure 5.13: Box plots of the accuracy (a) and Cohen's κ coefficient (b) of the cross validations on *best maximum memory cost* ordered by the corresponding performance metric (higher is better)

Table 5.9: Performance of the best learned prediction models with respect to *best maximum memory cost* for each prediction method on the *test set*; the classes "smart", "rt.leaf", and "mono." stand for *smart reduction*, *root leaf reduction*, and *monolithic minimisation*, respectively

Model	**Acc.**	κ	**Sensitivity**			**Specificity**			**Precision**		
			smart	**rt.leaf**	**mono.**	**smart**	**rt.leaf**	**mono.**	**smart**	**rt.leaf**	**mono.**
SVMwRK	**0.61**	**0.22**	0.85	0.24	0.30	0.39	0.87	**0.94**	0.67	**0.40**	**0.47**
SVMwLK	0.59	0.17	**0.87**	0.14	0.31	0.31	**0.91**	0.93	0.64	0.37	0.44
RF	0.60	**0.22**	0.83	0.23	0.33	0.43	0.86	0.92	0.68	0.37	0.42
LVQ	0.57	**0.22**	0.71	**0.36**	**0.34**	**0.55**	0.78	0.90	**0.70**	0.37	0.37
LDA	0.59	0.20	0.83	0.21	0.31	0.41	0.85	0.93	0.67	0.33	0.45
KNN	0.55	0.17	0.73	0.34	0.20	0.49	0.79	0.90	0.67	0.36	0.27
GLMER	0.58	0.16	0.84	0.14	0.30	0.39	0.86	0.92	0.66	0.26	0.40
CART	0.58	0.18	0.81	0.22	0.28	0.41	0.84	0.93	0.66	0.32	0.41

cross validations in learning the *best maximum memory cost*. The box plot shows the distribution of accuracy (Figure 5.13a) and Cohen's κ coefficient (Figure 5.13b).

The *no information rate* of the training data is 0.53; this indicates that any prediction model that has an accuracy above 53% performs better than a random guess based on the distribution of the data.

The highest median accuracy is 0.59 (achieved by SVMwRK, GLMER and RF). The highest median κ is 0.27 (RF). RF has both the highest maximum accuracy and κ at 0.67 and 0.37, respectively. The best prediction models among the cross validations have an accuracy between 0.58 and 0.67, and a κ between 0.27 and 0.42. Considering the no information rate, all of these best prediction models have a moderate accuracy and a weak inter-rater agreement.

Next, we investigate the performance of the best prediction models amongst the cross validations with respect to the training and test sets. Table 5.9 shows, for each prediction model, its *accuracy* (Acc. column) and κ coefficient, and its sensitivity, specificity, and precision with respect to the classes "smart" (for smart reduction), "root leaf" (for root leaf reduction), and "mono." (for monolithic minimisation). The best scores are indicated in bold.

The accuracy and κ on the training set are, respectively, 0.69 and 0.45 for SVMwRK, 0.60 and 0.29 for SVMwLK, 1.00 and 1.00 for RF, 0.62 and 0.36 for LVQ, 0.61 and 0.30 for LDA, 0.64 and 0.38 for KNN, 0.61 and 0.30 for GLMER, and 0.59 and 0.26 for CART (in the same order as the models appear in Table 5.9). All models, except for RF and KNN, are relatively stable in their accuracy; not differencing by more than 0.05 between training set and test set. All models have a significantly lower κ in the test set than in the training set. The least amount of change is seen for the models LDA and CART where the κ decreases from 0.30 to 0.20 and from 0.26 to 0.18, respectively. Despite that RF seems to have over fitted on the training set it still has a reasonably high performance on the unseen test set.

We shall now discuss the performance of the prediction models on the test set (see Table 5.9). The SVMwRK, SVMwLK, RF, and LVQ prediction models perform best on various aspects on the test set (see bold numbers in Table 5.9). However, amongst these the RF model is outmatched on all performance metrics but κ.

All prediction models have a very high *sensitivity* for the "smart" class, the lowest being 0.71 and the highest being 0.87. SVMwLK is the most sensitive to the "smart" class: it correctly classifies 87% of the cases that perform best with smart reduction. The "rt.leaf" and "mono." classes are predicted correctly at a low to moderate probability with sensitivity ranging from, respectively, 0.14 to 0.36, and 0.2 to 0.34. LVQ has the highest probability of predicting the "rt.leaf" and "mono." correctly: in respectively, 36% and 34% of the cases. As over half of the cases in the training set are classified as "smart", it may be the cases that there is not enough data to predict the other classes well.

Although the *specificity* for the "smart" class is slightly better for *best maximum memory cost* than for *best maximum number of transitions* it is still moderate: the probability of correctly predicting that a case should not be classified as "smart" ranges from 0.31 to 0.55. LVQ has the highest specificity to the "smart" class at 55%. Specificity is significantly better for the "rt.leaf" and the "mono." classes where specificity ranges from 0.78 to 0.91 for the root leaf reduction and from 0.90 to 0.94 for monolithic minimisation. Specificity for "rt.leaf" and for "mono." is the highest for SVMwLK and SVMwRK, respectively. The number of cases that do *not* belong to the "smart" class has increased with respect to the training set for the *best maximum number of transitions*. This seems to have resulted in a slightly better specificity for the "smart" class. Still the specificity is only moderate and may possibly be further improved by including more cases to the training set that are not of the "smart" class.

The precision of the "smart" class is high: the probability that a case predicted as "smart" is indeed a case belonging to the "smart" class ranges from 0.64 to 0.70 with LVQ having the highest precision. Precision for the "rt.leaf" class ranges from 0.26 to 0.40 and precision for the "mono." class ranges from 0.27 to 0.47, where SVMwRK achieves the highest precision for both classes.

Prediction on unseen cases is the most feasible for smart reduction. The sensitivity, and precision for the other classes is too low. Again we find indications that more data may improve sensitivity and precision for root leaf reduction and monolithic minimisation.

The five *most important* variables for the "smart" class were: *IM minimum*, *HM mean*, *IM product*, *IM median*, and *IM mean* (in descending order of importance). For the "rt.leaf" and "mono." classes the five most important variables were: *HM mean*, *IM product*, *IM mean*, *IM minimum*, and *IM median* (in descending order of importance).

5.7 Threats to validity

When interpreting the results of this study consider the following threats to validity:

- Only one tool has been involved to conduct the experiment, hence the results may be implementation specific. On the other hand, involving multiple tools introduces the problem that differences in implementations may affect the outcome.
- The scope of this study is limited to models that are represented as networks of LTSs. Therefore, the results of this study are possibly only applicable to models represented as networks of LTSs. As the compositional aggregation method is limited to these kind of models we have not considered alternative model representations.
- The study only considers the DPBB equivalence as aggregation relation. Results may vary depending on the chosen equivalence relation. The DPBB equivalence is the strongest aggregation order offered by CADP that still allows abstraction. Hence, other relations are expected to show better performance improvements.
- Models with a relatively small number of parallel processes were considered. Beyond models with six parallel LTSs the experiment quickly becomes infeasible. Extrapolation of the results presented in this work to models with more parallel LTSs should be done with caution. In the future, we plan to extend our analysis to subjects with more processes.
- The scaled up models, investigated in Section 5.5, make use of repeatable LTSs. It may be possible that the results are skewed due to lack of heterogeneous process LTSs. However, the used compositional aggregation methods do not take advantage of the symmetry in the model.

  The repeating of LTSs is noticeable in the violin plots (Fig. 5.5) as accumulation of sets of symmetric aggregation orders measuring the same normalized maximum number of transitions. Nevertheless, in most cases this effect does not change significantly as more repeated LTSs are added.
- Due to a bug in our aggregation order calculator the possible aggregation orders were not sampled completely uniformly at random. This affects RQ 5.1 and RQ 5.2. We were unable to retrieve results for these cases as our version of CADP was recently updated (which is more memory efficient) and we were unable to roll back. Hence, a fair comparison is impossible without redoing all experiments.

  After inspection, it turned out that about 3.8% of the orders were not considered in the subjects with six parallel processes. These orders represented a binary tree. Still, a portion of the aggregation orders forming a binary three were considered. The conclusions of our analysis are not strongly affected as the difference between compositional aggregation and monolithic minimisation and the samples and the heuristics are quite clear.
- In Section 5.5, a relatively small set of different cases has been studied, even though this experiment is the most comprehensive one performed thus far. The lack of a (publicly available) set of nicely scalable models is a problem in general when analysing and designing formal verification techniques.

- Despite our best efforts to generate realistic networks of LTSs in Section 5.6, the generated networks may still differ in many aspects from man-made models. If the generated networks that were used for training differ too much, then the learned predictors may not be applicable to man-made networks. Other aspects may have been missed due to distribution sampling of 88 source process LTSs, i.e., the variance of the set of source LTSs may have been too low. Still in this study we have shown that predictors for relevant response variables can be learned using machine learning.
- In the majority of the cases considered in Section 5.6 *smart reduction* performed best. As such, the data used to generated predictors for *best maximum number of transitions* and *best maximum memory cost* contained a significant class imbalance. We have dealt with this by using Cohen's κ coefficient as performance metric. Alternatives, such as up-sampling or down-sampling, may have given different results.

5.8 Conclusions

Our thorough analysis of compositional aggregation when applied to 119 subjects with varying topology, scale, and hiding set (Section 5.5) provides the following insights:

1. The amount of internal behaviour in process LTSs and the amount of synchronisation between process LTSs have the biggest impact on the performance, in terms of the largest number of generated transitions in memory.
2. The involvement of a functional property, and therefore a hiding set, is significant. The size of this hiding set is of less importance. For typical properties, maximal hiding already allows the hiding of a relatively large amount of behaviour.
3. Among the five network topologies we considered, none of them fundamentally rule out compositional aggregation as an effective technique.
4. As the number of processes in a model is increased, the effectiveness of compositional aggregation with respect to the monolithic approach tends to increase as well.

It should be noted that we only considered a few cases per topology. To generalise our conclusions, we will have to work on extending our benchmark set. The first two conclusions underline observations made in earlier work [56]. Since they worked with a set of subjects of less variety, we can make these observations with more confidence.

Machine learning on generated LTS networks showed that the maximum memory cost and maximum number of generated transitions of compositional aggregation heuristics normalised with respect to monolithic minimisation can be predicted at an average accuracy of one order of magnitude for half of the cases in our test set. The learned classification models that predict the best minimisation approach with respect to maximum memory cost and maximum number of generated transitions have an accuracy between 55% and 61% and between 68% and 74%, respectively, on unseen data. Furthermore, as both the training data and test data contain a class imbalance the Cohen's κ measure for inter-rate agreement is between 0.10 and 0.22, indicating that the models are better than random classification, but there is room for improvement. The classifiers were most sensitive to the class of cases for which smart reduction was the best choice with a sensitivity value of 0.87 for maximum memory cost and 0.99 for maximum number of

generated transitions. Metrics related to interleaving density and the number of transitions in an LTS were most important for regression techniques, while metrics related to hiding and interleaving density of sets of LTSs were found to be the most determining factor for classification techniques.

Future Work In the near future, we will extend the current analysis to further explain the success and failure of compositional aggregation for the different subjects, and based on this, work on the construction of a new heuristic. For this to be successful, we will have to involve many more cases. As scalable models have now been thoroughly investigated, we can next focus on non-scalable models, of which many are publicly available.

To strengthen the validation in this chapter, we will validate the results against a number of LTS networks provided by the literature.

The generation of realistic LTS networks also has many interesting directions for future work. We would like to investigate the use of commonly occurring graphlets in the generation of LTSs. Furthermore, more intelligent ways to generate the LTS networks may be considered. For instance, our generator has a small chance of generating a model that has no reachable states other than the initial state; we see potential in an alternative method where an unlabelled state space is constructed before labelling transitions of LTSs to ensure reachability of nearly all states in process LTSs.

We had to aggregate some of the metrics before we could apply machine learning techniques. The ability to apply machine learning techniques to graphs directly would open up many opportunities by preventing the need to aggregate graph data.

Chapter 6

Avoidance of Sequential Consistency Violations under Relaxed-Memory Models

When targeting modern hardware architectures with parallel computation capabilities, constructing software that is correct and optimally takes advantage of these capabilities is complex and time-consuming. In particular, interleaving orders that break intended atomic behaviour are a major source of bugs. However, applying synchronisation mechanisms to repair atomicity should be done sparingly: synchronisation mechanisms can cause contention negatively impacting performance.

This chapter develops a monitor that detects sequential consistency violations in concurrent programs specifying atomic behaviour. The monitor is optimised to reduce its memory footprint by summarising the program order of the trace it monitors. A further optimisation step is proposed that makes the monitor efficient for detection of sequential consistency violations in model checkers. The model checker only needs to considering sequences of atomic instructions, thereby, avoiding a significant amount of interleaving and reordering of memory accesses. Moreover, the monitor remembers the information that is relevant to suggest a minimal set of locks and delays (e.g., memory fences).

The monitor over approximates certain aspects of the program while it is more precise than static analysis in other aspects as it considers the dynamic semantics of the program. Through a pre-analysis (e.g., one reporting only critical cycles) the monitor can be focussed on a given set of accesses to reduce over-approximation of the cycle detection while maintaining the precision gained from evaluation of the dynamic semantics of the program.

The proposed monitor is built on the well known theory of Shasha and Snir, and therefore, supports weak memory that guarantee store atomicity.

This chapter has not yet been published elsewhere

de Putter, S., and Wijs, A. Model Driven Avoidance of Atomicity Violations under Relaxed-Memory Models. In *ESOP* (2019). *Submitted*

6.1 Introduction

When developing parallel software it is very challenging to guarantee the absence of bugs. Achieving the intended execution order of instructions while obtaining a high performance is extremely hard. In particular, interleaving orders that break intended sequential and atomic behaviour are a major source of bugs [140]. These can be avoided by appropriately using synchronisation mechanisms such as fences, semaphores, hardware-level atomic operations, and software/hardware transactional memory [189]. On the other hand, over using such mechanisms can cause contention, which negatively impacts performance and therefore defeats the purpose of using parallelism in the first place. When using transactional memory, Hardware Transactional Memory (HTM) is preferable over software support [67], for performance reasons, but the former cannot handle transactions of arbitrary size. Even if the transactions are defined small enough, HTM can still negatively impact performance, which is in part influenced by the size of the transactions [27]. In other words, synchronisation mechanisms should be used only for those memory accesses that may be involved in a violation of the expected behaviour.

Sequential consistency is arguably the most understood concurrency model. An (execution) trace of a concurrent program is *Sequentially Consistent* (SC) iff all memory accesses are performed in program order, atomicity constraints are respected, and accesses are serviced from a single First In First Out (FIFO) queue. As this model does not deviate from the software developers' specification it is the most intuitive programming model.

Sequential consistency is a very restrictive model and does not benefit from modern compiler and processor optimisations. It is sufficient, however, that traces of a program produce results that are *observably* equivalent to SC traces. Such traces are said to be SC serialisable. A trace is *SC serialisable* iff it can be rewritten into an SC trace without commuting conflicting accesses. Two accesses are in *conflict* iff they access the same (memory) location and at least one of them is a write access.

Shasha and Snir proposed a method to derive a minimal set of locks and delays that together restrict the possible executions to SC serialisable ones [188]. The *set of locks* specifies which accesses may be mutually exclusively executed while the *set of delays* enforces a partial order on accesses, indicating which accesses must be delayed until others have finished. Sasha and Snir use a specification of the program to derive a dependency graph. The locks and delay sets are derived from so-called critical cycles that are found in this dependency graph. To achieve their static dependency analysis, techniques must be used that are quite pessimistic as they must be conservative [5]. Such analyses is particularly pessimistic in the presence of program branches: certain executions may in fact not be reachable due to branches being disabled at run-time.

Sequential consistency monitoring is an alternative approach that is more precise with respect to branches. A monitor analyses executions as streams of memory accesses and can, thus, be applied at run-time. Similar to Shasha and Snir a dependency graph between these accesses is built and a cycle indicates a sequential consistency violation. These dependency graphs are called *conflict graphs* [19, 39, 159]. To derive locks and delays, a model checker can be applied to feed a trace to a monitor [73]. While this is more precise with respect to branches, a monitor inherently reports more than just the critical cycles.

In this chapter, we consider concurrent programs consisting of a number of threads, each performing a number of atomic instructions. Each instruction can perform one or more memory accesses.

We propose a sequential consistency monitor that combines conflict graphs and the theory of Shasha and Snir. On the one hand, the monitor over approximates certain aspects of the program since it finds all cycles (not just the critical ones). On the other hand, the monitor is more precise as it considers the dynamic semantics of the program. The efficiency of the monitor can be improved by focusing its inspection to accesses reported by a pre-analysis algorithm (e.g., the algorithm of Shasha and Snir). This increases both the precision and the run-time performance of the monitor.

Furthermore, we propose a summarised conflict graph that summarises completed instructions. Finally, the summarised conflict graph is optimised further for the use in a model checker to efficiently derive locks and delays similar to Shasha and Snir. By building on the theory of Shasha and Snir our results cover many weak memory models such as Total Store Order, Release Consistency, Partial Store Order, and Relaxed Memory Order.

In the literature little attention is given to preservation of sequential consistency for Domain Specific Languages (DSLs) when generating code. To this end, the proposed conflict graph is applied in the mCRL2 [54] model checker to derive a minimal sets of locks and delays for a number of programs specified in the Simple Language of Communicating Objects (Slco) [71]. These locks and delays ensure observably SC behaviour with respect to the Slco semantics.

Contributions We propose a monitor that detects sequential consistency violations. The correctness of the monitor is deduced from the theory of Shasha and Snir; thereby achieving support for a wide variety of weak memory models.

We show that completed instructions can safely be summarised if all the preceding instructions are completed. This leads to a summarised conflict graph that is more memory efficient.

The monitor is further optimised for the use in model checkers. The monitor treats SC traces of instructions. In contrast to related literature, the instructions are treated as a whole, thereby reducing the number of interleaving accesses that need to be explored. Furthermore, we summarise the conflict relation between instructions, eventually resulting in self-loops on vertices of the conflict graph in case those vertices are involved in a sequential consistency violation, which manifests itself in the graph as a loop.

To further improve efficiency of violation detection, we filter out accesses that are deemed safe by a static analysis: these safe accesses are ignored by the conflict graphs and, furthermore, allow the application of Partial Order Reduction (POR) in the model checker. We employ the approach in the mCRL2 model checker on a number of experiments for the Slco DSL.

Structure of the chapter. Section 6.2 briefly reviews related work. In Section 6.3, the theory of Shasha and Snir is introduced. We explain the shortcomings of existing conflict graphs and summarised conflict graphs, and propose improvements for monitoring SC in Section 6.4. Section 6.5 proposes further summarised conflict graph optimisations specifically tailored for deriving lock and delay sets with a model checker. An implementation of sequential consistency checking with a model checker is discussed in Section 6.6, and experimental results are presented in Section 6.7. Finally, conclusions and pointers to future work are given in Section 6.8.

6.2 Related Work

The monitor proposed in this chapter uses a conflict graph that is inspired by database and atomicity monitoring literature [19,39,73]. Farzan and Madhusudan employ a summarised conflict graph, where completed accesses are summarised to reduce memory cost, to detect atomicity violations using a model checker [73]. They assume that accesses performed by a thread are executed in program order. This is a shortcoming, as modern hardware gives no such guarantees.

We show that the conflict graph and summarised conflict graph proposed by Farzan and Mudhusudan indeed do not detect all atomicity (and sequential consistency) violations when compilers and processors are allowed to reorder accesses. Furthermore, we present an improved conflict graph and a summarised conflict graph that monitor sequential consistency (which is strictly stronger than atomicity). In addition, we present an adapted summarised conflict graph that is further optimised for model checkers. Using this monitor it suffices to consider all SC instruction traces instead of all SC access traces, thus, a significant number of interleaving is avoided.

There is more related work in the field of valid-time databases [41, 75]. In valid-time databases transactions must be serialised in submission order. However, here the source of the atomic transaction (database equivalent to an instruction) is not considered. Therefore, consistency from the point of view of the client (similar to a program thread) is not considered.

Burnim, Sen, and Stergiou propose a sequential consistency monitor for Total Store Order and Partial Store Order architectures [38]. They summarise the dependency relation between accesses by modelling the store buffer. By encoding the buffer semantics explicitly the monitors are able to summarise more behaviour than the monitor presented in this chapter. As their monitors are defined on access traces, their monitors would require similar optimisations as ours to efficiently be employed in a model checker. Additionally, our algorithm also allows derivation of sets of locks and delays that guarantee sequential consistency.

With respect to avoiding locks where possible, transforming code to remove locks [227] is related, although we aim to avoid locking initially. Poetzl and Kroening [172] transform threads to optimise them. It would be interesting to use their results to further refine produced code. In work on model-based code generation [42, 99, 175] automatic optimisations are being applied. However, these do not analyse the model to produce a tailor-made implementation. In [66], a state machine-based language is presented together with a data race checker, but sequential consistency checking is not addressed.

Kuperstein, Vecheve, and Yahav [128] propose a model checking algorithm to automatically compute a set of memory fences that restricts a program P to SC-serialisable traces under a given memory model M. A downside of their approach is that they compute the transition system of the given program under M involving interleaving and possible reordering of accesses. In contrast to their approach, our algorithm only needs to consider SC traces. As our method does not target specific memory models it would be interesting to see if the two approaches could be combined.

6.3 Guaranteeing Sequential Consistency

Sequential Consistency of a Program. To reason about sequentially consistent behaviour of a concurrent program we consider its execution on, and its effects on the

memory of, a multi-core machine. To this end we assume that the state of the machine is defined by the values stored in its storage locations. The set of storage *locations* of a machine is denoted by $\mathbb{L}$. A location may be a register, a memory location (e.g., associated to some variable), or another medium.

Figure 6.1 gives an overview of the concepts related to a concurrent program used in this work. A program consists of a set of threads $\mathbb{T}$, each performing a sequence of atomic instructions called a *thread program*. An atomic *instruction* executes one or more operations atomically. Each *operation* performs zero or more *accesses* to locations.

An *access* either reads from, or writes to, a given location $\ell \in \mathbb{L}$ and has a unique identifier. We write $r_t(\ell)$ and $w_t(\ell)$ for read and write accesses, respectively, by a thread $t \in \mathbb{T}$ at a location $\ell \in \mathbb{L}$ (leaving the identifier implicit). An access is performed *atomically*, i.e., two accesses on the same location behave as if they occur serially in some order. The set of accesses of a thread $t \in \mathbb{T}$ is denoted by $\mathbb{V}_t$. The set of all accesses of a program is defined as $\mathbb{V} = \bigcup_{t \in \mathbb{T}} \mathbb{V}_t$.

The order of the thread program $t \in \mathbb{T}$, or the *thread order* for short, is defined by an irreflexive total order $P_t \subseteq \mathbb{V}_t \times \mathbb{V}_t$ on the accesses performed by t. The *program order* $P \subseteq \mathbb{V} \times \mathbb{V}$ consists (at least) of the union of the thread orders of all threads: $\bigcup_{t \in \mathbb{T}} P_t \subseteq P$. In most cases, accesses of different threads are unrelated by P. However, P may relate accesses of different threads to represent, for instance, synchronisation constraints across threads.

The atomic instructions of some thread $t \in \mathbb{T}$ are defined using an equivalence relation $A_t \subseteq \mathbb{V}_t \times \mathbb{V}_t$ identifying classes of accesses that are to be performed (observably) atomically. We write $[a]$ for the equivalence class under A_t that contains access $a \in \mathbb{V}_t$ plus any other accesses in $\mathbb{V}_t$ that are equivalent to a according to A_t. The atomicity relation over all accesses of a program is $A = \bigcup_{t \in \mathbb{T}} A_t$. In examples, we write $\langle \ldots \rangle$ to indicate an atomic instruction, i.e., all accesses within the brackets are related by A.

The quotient order X/A of an order $X \subseteq \mathbb{V} \times \mathbb{V}$ aggregates X by the atomic instructions of A: $[a_1]\ X/A\ [a_2]$ iff $[a_1] \neq [a_2]$ and there exist $a \in [a_1]$, $a' \in [a_2]$ such that $a\ X\ a'$. The quotient order P/A is the program order over atomic sets and, hence, defines a partial order over instructions.

A relation R can be *extended* to a total order iff its transitive closure is irreflexive. Consider a graph representing R. It follows that R can be *extended* to a total order iff this graph is acyclic.

An *execution trace*, or *trace* for short, is a sequence of accesses ordered by a total *execution order* $\prec$. A trace captures an (interleaved) execution of a multi-threaded program and describes the order in which the accesses of the program segments of threads are (visibly) performed.

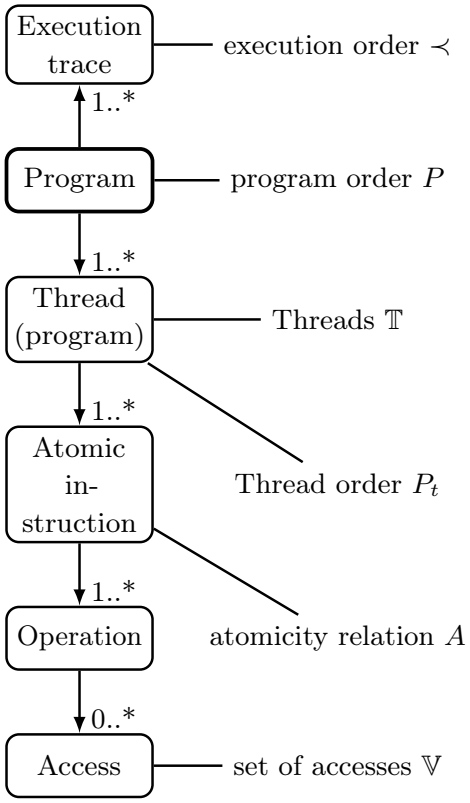


Figure 6.1: Overview of a program

Programmers *rely* on a programming paradigm where atomic instructions are, or appear to be, executed in programmed order and without interruption. That is, programmers

expect their program to behave *sequentially consistent*. Hence, a program is *correct* if all possible execution traces appear to be sequentially consistent (SC).

Definition 6.3.1 (Sequential Consistency)**.** *An execution trace π is* sequentially consistent (SC) *iff π is an extension of P in which for each instruction, its accesses appear in an uninterrupted sequence, i.e., π is SC iff it extends P and is of the form $S_1 \prec S_2 \prec \ldots \prec S_n$ with n the number of instructions (i.e., $n = |\mathbb{V}/A|$) and S_i ($i \in 1..n$) a sequence of accesses of an instruction.*

An SC trace executes all atomic instructions sequentially, without interruption, following the program order. Hence, by definition any SC trace retains atomicity of instructions. It is sufficient, however, that a trace is observably no different from an SC trace.

SC up to observation. A trace appears to be sequentially consistent if it computes the same values as an SC trace. The observable values of some location $\ell \in \mathbb{L}$ are restricted by the order in which reads from and writes to ℓ occur. The potential values of ℓ may depend on the order in which two accesses on ℓ are executed if one of these accesses is a write access. In this case, those accesses are said to be in conflict.

Definition 6.3.2 (Conflict)**.** *Two accesses* conflict *iff they access the same location $\ell \in \mathbb{L}$ and at least one of them is a write access.*

Two traces π, π' are *equivalent* if they compute the same values, meaning that they read and write the same values. As long as one is a permutation of the other and conflicting accesses appear in π and π' in the same order, the values written to and read from locations remain the same.

Definition 6.3.3 (Trace Equivalence)**.** *Two traces π and π' are* equivalent*, denoted $\pi \approx \pi'$ iff π can be rewritten to π' (and vice versa) by commuting non-conflicting accesses.*

A trace π is said to be *SC-serialisable* if it is equivalent to an SC trace.

Definition 6.3.4 (SC-Serialisable)**.** *An execution trace π is* SC-serialisable *iff there exists an SC trace π' such that $\pi \approx \pi'$.*

From the perspective of a programmer a program behaves *correctly* iff all possible execution traces are *SC-serialisable*.

Detecting non-SC-serialisable traces. A non-SC-serialisable trace (or non-SC trace for short) exists when accesses of different threads have a cyclic conflict with each other. To formally detect such cyclic conflicts a conflict relation C is used. The *conflict relation* $C : \mathbb{V} \times \mathbb{V}$ is an irreflexive symmetric relation relating any two accesses (of different threads) that are in conflict. An orientation O of C is an irreflexive asymmetric relation specifying the order in which conflicting accesses are to be executed; $a_1\ C\ a_2$ iff either $a_1\ O\ a_2$ or $a_2\ O\ a_1$.

Violation of SC-serialisability can be detected by searching for cycles in $C/A \cup P/A$ as indicated by Proposition 6.3.5. We assume that conflicts within an instruction are not commuted (by compilers and processing units) as this is always unsound.

Proposition 6.3.5. *A program has non-SC-serialisable traces iff $C/A \cup P/A$ contains a cycle. Furthermore, if O is an orientation of C the class of traces defined by O is non-SC-serialisable iff $O/A \cup P/A$ contains a cycle.*

Proof. We have assumed that conflicting accesses are not commuted within atomic instructions. It follows from Lemma 2.1 of Shasha and Snir [188] that a program has non-SC traces iff there exists an orientation O of C such that $O/A \cup P/A$ contains a cycle. It follows that the program has non-SC traces iff $C/A \cup P/A$ contains a cycle. □

If there is a cycle in $C/A \cup P/A$ a non-SC-serialisable trace can be performed by executing the accesses of any non-singleton atomic set of accesses in the cycle and interleaving between these the other accesses in the cycle in a certain order. Figure 6.2 shows such a cycle causing a violation of sequential consistency. Edges marked by P and C indicate P-edges and C-edges, respectively. The grey boxes indicate atomic instructions. A *solid* line or box indicates a single edge or instruction, respectively. A *dashed* line or box indicates a path of zero or more edges or instructions, respectively. For instance, a trace $\pi = a_1 \prec a_{n-1} \prec a_n \prec \ldots \prec a_{i-1} \prec a_i \prec \ldots \prec a_3 \prec a_4 \prec a_2$ is not SC-serialisable. It is constructed by respecting the P-order within each involved instruction, and placing the instructions against the cycle direction between accesses a_1 and a_2. Although for all *even* i $(4 \leq i \leq n)$ access a_i may be commuted towards the right, a_{i-1} cannot follow because of its conflict with a_{i-2} (and symmetrically for *odd* i $(3 \leq i < n)$). Hence, a_1 and a_2 cannot be brought together without breaking atomicity or program order of other instructions. In addition, compilers and processing units may (locally) reschedule non-conflicting accesses leading to other possible non-SC traces. For instance, if a_1 and a_2 do not conflict with each other, another non-SC trace can be obtained by completely sticking to the cycle order, but swapping a_1 and a_2.

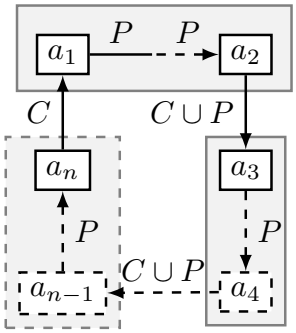


Figure 6.2: A generic cycle in $C/A \cup P/A$

Example 6.3.6 (A non-SC program)**.** *Consider the following program:* $t_1 = \langle w_1(y)\ P_1\ r_1(x)\rangle$ *and* $t_2 = \langle w_2(x)\rangle\ P_2\ \langle r_2(y)\rangle$*. As compilers and processors may (locally) reschedule non-conflicting accesses the non-SC trace* $\pi = r_1(x) \prec w_2(x) \prec r_2(y) \prec w_1(y)$ *may occur. This trace contains the following cycle:* $r_1(x)\ C\ w_2(x)\ P_2\ r_2(y)\ C\ w_1(y)\ A\ w_1(x)$*.*

In π *atomicity of* $\langle w_1(y)\ P_1\ r_1(x)\rangle$ *is broken and the accesses appear out of order. To reach an equivalent trace in which atomicity is not broken and the accesses of* t_1 *are in order our only option is to, first, commute the two accesses of* t_2*, and then commute the accesses of* t_1 *inward:* $\pi = r_1(x) \prec w_2(x) \prec r_2(y) \prec w_1(y) \approx r_1(x) \prec r_2(y) \prec w_2(x) \prec w_1(y) \approx r_2(y) \prec w_1(y) \prec r_1(x) \prec w_2(x) = \pi'$*. However, by commuting to* π' *we are forced to break the thread order of* t_2*. Hence, the program is not SC.*

Guaranteeing SC-serialisability. To guarantee correctness of a program all possible SC violations have to be ruled out. As these are represented by cycles in $C/A \cup P/A$, we have to apply appropriate synchronisation constructs to the accesses in such cycles. Next, we discuss how to do this.

The *delay* relation $D \subseteq P \cup A$ is a partial order that enforces a (local) precedence relation between accesses of a program. If $a_1\ D\ a_2$, then a_2 is delayed until access a_1 is executed *and* a_2's commit to memory is delayed until a_1 is committed to memory. A delay between two accesses may be achieved by applying a memory fence.

Definition 6.3.7 (Sufficient Delay Relation)**.** *A delay relation* D *is* sufficient *iff all execution traces respecting* D *are SC-serialisable with respect to* P*.*

A delay relation D is *minimal* iff it is sufficient and there is no strict subset $D' \subset D$ that is sufficient. SC-violations can be found by detecting *simple* cycles in $C/A \cup P/A$. A *simple cycle* may contain any node and edge at most once; edges from A and C are considered non-directional edges (e.g., a simple cycle may contain either $a_1\ C\ a_2$ or $a_2\ C\ a_1$, but never both). A minimal delay relation can be computed by detecting *critical cycles* in $C/A \cup P/A$.

Definition 6.3.8 (Critical Cycle)**.** *A cycle σ in $C/A \cup P/A$ is a* critical cycle *if it is a simple cycle and has no chords in P/A (i.e., $(a_1, a_2) \notin P/A$ for any two non-adjacent accesses a_1 and a_2 in the critical cycle).*

Because of the absence of chords in P/A, a critical cycle is a minimal cycle with no more than two successive accesses in any thread.

A minimal delay relation that guarantees sequential consistency of a program can be derived from the $P \cup (A \setminus C)$-edges represented in the set of all critical cycles of the program.

Proposition 6.3.9 ([188])**.** *Let D_0 consist of all edges in $P \cup (A \setminus C)$ that are represented by a critical cycle in $C/A \cup P/A$. Then, D_0 is sufficient iff D_0 is acyclic. Furthermore, D_0 is minimal iff D_0 is acyclic.*

In contrast to Shasha and Snir we do not include A-edges that are also C-edges in the definition of a critical cycle. Otherwise, say that $(a_1, a_2) \in P$ and that $(a_2, a_1) \in A \cap C$ is in a critical cycle, then it is possible to include (a_2, a_1) in D_0, thus enforcing a delay against program order between accesses that are in conflict. In this case D_0 becomes cyclic as the reverse edge $(a_1, a_2) \in P$ in the direction of the program order is also contained in a critical cycle. The exclusion of $A \cap C$-edges in critical cycles does not influence the proofs presented by Shasha and Snir.

If D_0 is acyclic, then the relation D_0 specifies a minimal set of constraints that limits a program to SC-serialisable executions. When D_0 contains a cycle, a sufficient delay relation does not exist. The accesses in these delay cycles must be protected by a synchronisation mechanism that prevents mutual access to locations (e.g., through the use of locks or transactional memory).

We use an equivalence relation $M \subseteq A$ to define classes of accesses that require prevention of mutual accesses, and a locking protocol to achieve this prevention. For each class $l \in M$ the protocol protects all locations accessed by the accesses in l. The protection starts at the first executed access in l and ends when all accesses in l have been executed. Multiple read accesses may read from the same location while the location is not being written to. At all times only one access may have access to a location that is being written to.

Shasha and Snir propose a minimal locking relation $M \subseteq A$ and derive a minimal delay relation $D(M) \subseteq P \cup (A \setminus C)$ that guarantees SC-serialisable execution of a program:

Proposition 6.3.10. *Let M be defined as follows: $a_1\ M\ a_2$ iff a_1 and a_2 are in the same strongly connected component of the graph representing D_0. Furthermore, let $D(M) = (D_0 \setminus M) \subseteq P \cup A$. Then,*

1. *$M \subseteq A$, $D(M) \subseteq P \cup A$, and $D(M)$ and $D(M)/M$ are acyclic;*
2. *M and $D(M)$ ensure SC-serialisability of execution traces (constrained by M and $D(M)$); and*

3. M is minimal, and $D(M)$ is minimal (after fixing M).

Proof. We have assumed that conflicting accesses are not commuted within atomic instructions. Therefore, we may omit delays reported by the work of Shasha and Snir that are within instructions. The remainder of the proof follows from Theorem 5.4 of Shasha and Snir [188]. □

We briefly discuss the conclusions of the proposition above. The *first* point states that M and $D(M)$ are subsets of the intended relations and that the delay relation $D(M)$ does not contain any cycles that could prevent it from being applied. The *second* point says that if the program is constrained by M and $D(M)$, then all traces of the program are SC-serialisable. Finally, the *third* point indicates that there are no smaller sets M' and D' that together are sufficient to guarantee SC-serialisability of all traces. Note that the last point only holds for programs that do not have branches (e.g., *if-then-else* statements) as the theory so far has not considered these.

6.4 Monitoring Conflict Serialisability Violations

Conflict Graphs. The use of a conflict graph or a summary of it is common practice to monitor transaction traces in the database literature [19, 39, 159]. While this literature is focussed on monitoring atomicity violations, there is an increasing interest in monitoring sequential consistency violations [38, 159]. In the field of databases a *transaction* corresponds to an atomic *instruction* and an *event* corresponds to an *access* or a symbol denoting the completion of a transaction. However, for the sake of consistency we will keep using the terms introduced in the previous section.

In the remainder an access a is an element of a trace π, denoted by $a \in \pi$, iff a occurs in π. The *projection of a set* S of accesses onto a trace π is define as $S^\pi = \{a \mid a \in S \wedge a \in \pi\}$. Similarly, the *projection of a relation* $R : \mathbb{V} \times \mathbb{V}$ onto a trace π is define by $a_1 \; R^\pi \; a_2$ iff $a_1 \; R \; a_2$ and $a_1, a_2 \in \pi$.

Accesses of a thread t have a unique identifier based on their rank in P_t, i.e., the thread program execution order. Hence, also instructions have a unique identifier. Let I_t be the set of instruction identifiers, and let $I = \bigcup_{t \in \mathbb{T}} I_t$. The set of instruction identifiers of t that are in π (completed or uncompleted) is defined by I_t^π. The set of all instructions in π is defined by $I^\pi = \bigcup_{t \in \mathbb{T}} I_t^\pi$. In the remainder, we write $[i]$ to refer to the set of accesses of an instruction i. We say that i is P-before an instruction $i' \in I^\pi$ in a trace π iff $[i] \; P/A \; [i']$, meaning that at least one access of i is P-before at least one access of i'.

Furthermore, there is a special trace element $\checkmark_i \in \pi$ that denotes the *completion* of an instruction $i \in I^\pi$. The atomicity relation A^π can be derived from a trace involving this special element if the relation is not known otherwise.

A conflict graph is a tool for monitoring execution traces, and is therefore defined by a trace π. Furthermore, since it monitors sequential consistency of a trace the orientation of conflicts is relevant, i.e., we do not want to report a violation when an observed trace is SC (even if there exist variations that are not SC). A trace violates *atomicity* constraints iff its conflict graph contains a cycle [73].

Definition 6.4.1 (Conflict Graph [73,159])**.** *The* conflict graph *of a trace π is the directed graph $G_\pi = (V, E, L)$. The set V consists of vertices representing instructions $i \in I^\pi$. L is a vertex-labelling function, labelling a vertex representing $i \in I^\pi$ with $[i]^\pi$, i.e., the set*

of accesses of i that are in π. The set E consists of edges. There is an edge $(i_1, i_2) \in E$ between two vertices/ instructions $i_1, i_2 \in V$ iff

- $i_1, i_2 \in I_t^\pi$ *of some thread t and* $L(i_1) \prec/A^\pi\ L(i_2)$, *or*
- $L(i_1)\ C^\pi/A^\pi\ L(i_2)$ *and* $L(i_1) \prec/A^\pi\ L(i_2)$.

A trace π violates *atomicity* observably when there is a cycle in the conflict graph G_π [73]. However, the current conflict graph does not consider thread local optimisations, that is, compilers or processing units may reorder accesses within thread programs that are not conflicting. Example 6.4.2 shows that there are non-SC traces with acyclic conflict graphs.

Example 6.4.2 (A non-SC trace missed by its conflict graph)**.** *Reconsider the non-SC trace of Example 6.3.6 $\pi = r_1(x) \prec w_2(x) \prec \checkmark_{\langle w_2(x)\rangle} \prec r_2(y) \prec w_1(y)$ presented in Example 6.3.6 and the corresponding program: $t_1 = \langle w_1(y)\ P_1\ r_1(x)\rangle$ and $t_2 = \langle w_2(x)\rangle\ P_2\ \langle r_2(y)\rangle$. Figure 6.3 presents the conflict graph corresponding to the non-SC trace. Although the trace is non-SC, the violation is not observed in G_π, since there is no directed cycle. To detect the SC-violation an edge from vertex $\langle r_2(y)\rangle$ to vertex $\langle w_2(x)\rangle$ is needed.*

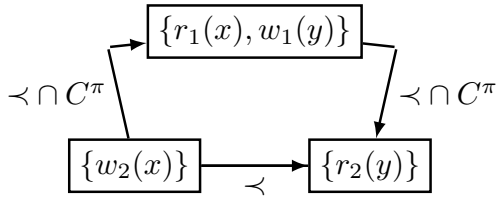


Figure 6.3: Conflict graph G_π with $\pi = r_1(x) \prec w_2(x) \prec \checkmark_{\langle w_2(x)\rangle} \prec r_2(y) \prec w_1(y)$

An Improved Conflict Graph. To add support for SC-violations due to thread local reordering we adapt the conflict graph to the theory of Shasha and Snir. For this, the conflict graph of a trace π requires knowledge about the local thread orders for each thread $t \in \mathbb{T}$ involved in π. In the area of serialisability of database transactions this can be achieved by supplying the transaction rank upon submission of the transaction. In the area of sequential consistency of concurrent programs the conflict graph needs the relation P_t^π. The specified order for π is defined as P^π.

Definition 6.4.3 (Improved Conflict Graph)**.** *The* improved conflict graph *of a trace π is the directed graph $IG_\pi = (V, E, L)$. Sets V, L are defined as for G_π. The set E consists of edges. There is an edge $(i_1, i_2) \in E$ between two vertices/ instructions $i_1, i_2 \in V$ iff*

1. $i_1, i_2 \in I_t^\pi$ *of some thread t and* $L(i_1)\ P_t^\pi/A_t^\pi\ L(i_2)$, *or*
2. $L(i_1)\ C^\pi/A^\pi\ L(i_2)$ *and* $L(i_1) \prec/A^\pi\ L(i_2)$.

Reconsider Example 6.4.2. In IG_π, an edge is drawn from $\{r_2(y)\}$ to $\{w_2(x)\}$ (based on the thread program order P_t) instead of in the other direction, by which the ordering $\prec$ of π for accesses within the second thread is ignored and the presence of a sequential consistency violation is revealed.

Propositions 6.4.4 and 6.4.5 show that IG_π contains a cycle iff π is a non-SC trace.

Proposition 6.4.4. *Consider a trace π. If IG_π contains a cycle, then π is a non-SC trace.*

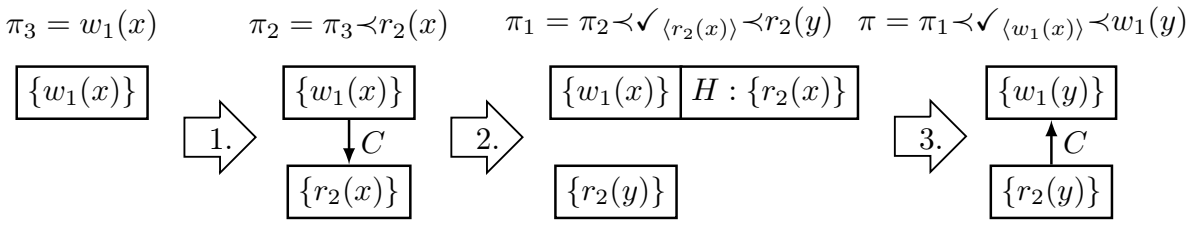


Figure 6.4: Construction of the summarised conflict graph for a non-SC trace $\pi = w_1(x) \prec r_2(x) \prec \checkmark_{\langle r_2(x)\rangle} \prec r_2(y) \prec \checkmark_{\langle w_1(x)\rangle} \prec w_1(y)$ with program $t_1 = \langle w_1(y)\rangle\ P_1\ \langle w_1(x)\rangle$ and $t_2 = \langle r_2(x)\rangle\ P_2\ \langle r_2(y)\rangle$ according to Farzan and Madhusudan [73].

Proof. The edges of IG_π are defined by 1) $\bigcup_{t\in\mathbb{T}} P_t^\pi/A_t^\pi$ and 2) $C/A^\pi \cap \prec/A^\pi$. For the *first* set we have $\bigcup_{t\in\mathbb{T}} P_t^\pi/A_t^\pi = P^\pi/A^\pi$. Furthermore, the *second* set contains C/A^π. Hence, if there is a cycle σ in IG_π, then σ is a cycle in $C/A^\pi \cup P^\pi/A^\pi$. It follows from Proposition 6.3.5 that π violates sequential consistency. □

Proposition 6.4.5. *Let π be a non-SC trace, then IG_π contains a cycle.*

Proof. Let $O^\pi \subseteq \prec \cap C^\pi$ be the orientation of C^π followed by $\prec$. Since π is a non-SC trace, then by Proposition 6.3.5, there is a non-trivial cycle σ in $O^\pi/A \cup P/A$ that only involves accesses in π. Furthermore, $O^\pi \subseteq \prec \cap C^\pi$ and $\bigcup_{t\in\mathbb{T}} P_t^\pi/A_t^\pi = P^\pi/A^\pi$ by definition. Hence, σ is also a cycle of IG_π. □

Note that for a trace π, IG_π contains *all* cycles, including non-critical ones.

Summarising the Conflict Graph. Maintaining vertices for each instruction in a stream of instructions can impose significant burden on a machine's memory. For this reason summarised conflict graphs [39] were introduced. In a summarised graph each vertex corresponds to a thread $t \in \mathbb{T}$ and contains, besides the current instruction, an aggregation of t's completed instructions and conflicting accesses of other threads.

The summarised conflict graph of Farzan and Madhusudan [73] uses a second vertex-labelling function H that collects, for any thread t and instruction i of t, the accesses of completed instructions of other threads that had conflicts with i. Whenever an instruction i is completed, the vertex corresponding to i is removed from the summarised conflict graph, all the (immediate) predecessors and successors of the vertex are connected, and the function H is updated. However, this summary loses essential information to detect that a trace violates the intended observable order P as shown in the following example.

Example 6.4.6 (The summarised conflict graph of Farzan and Madhusudan [73] may lose information)**.** *Consider the program $t_1 = \langle w_1(x)\rangle\ P_1\ \langle w_1(y)\rangle$ and $t_2 = \langle r_2(y)\rangle\ P_2\ \langle r_2(x)\rangle$, and corresponding non-SC trace $\pi = r_2(x) \prec \checkmark_{\langle r_2(x)\rangle} \prec w_1(x)$. Once the first access in the trace has been executed, a vertex is added to the graph for $\langle r_2(x)\rangle$. Reading the second element, i.e., $\checkmark_{\langle r_2(x)\rangle}$, this vertex is deleted again. Since this vertex has no successors or predecessors, the fact that $r_2(x)$ took place is lost. Naturally, as the execution continues, the summarised conflict graph will never contain a cycle.*

In addition to the flaw above, essential information may be lost when instructions are not immediately completed.

Example 6.4.7 (The summary function H may lose information when access order is relevant)**.** *Consider the same program as in Example 6.4.6, and non-SC trace $\pi =$*

$w_1(x) \prec r_2(x) \prec \checkmark_{\langle w_1(x)\rangle} \prec w_1(y) \prec \checkmark_{\langle r_2(x)\rangle} \prec r_2(y)$. *Figure 6.4 shows how the summarised conflict graph for* π *is constructed:*

1. *A new vertex for* $\langle r_2(x)\rangle$ *is introduced and an edge is drawn for the conflict from the vertex for* $\langle w_1(x)\rangle$ *to the new vertex.*
2. *A summary is accumulated when* $\checkmark_{\langle w_1(x)\rangle}$ *appears; the vertex for* $\langle w_1(x)\rangle$ *is deleted, and its access is added to* $H(\langle w_1(x)\rangle)$*. Furthermore, a vertex is created for* $\langle w_1(y)\rangle$*.*
3. *When* $\checkmark_{\langle w_1(x)\rangle}$ *appears, the vertex* $\langle w_1(x)\rangle$ *is deleted along with the information it contained via* H*. Hence, in the final graph, no cycle is present; the information required to form a cycle, i.e., that* t_1 *executed a write to* x*, has been forgotten.*

From the theory of Shasha and Snir it is clear that accesses that are P-before a currently considered access may still cause sequential consistency violations. It follows that accesses performed by a thread must be remembered. In the summarised conflict graph presented below (Definition 6.4.8) such accesses are stored in special vertices called *P-summaries*, one for each thread $t \in \mathbb{T}$, denoted by PS_t. A P-summary PS_t summarises all past accesses of t, thus maintaining the transitive closure of P_t. We refer with PS to the set of all the PS_t: $PS = \{PS_t \mid t \in \mathbb{T}\}$.

In particular, given an access, PS_t stores the location and type of access performed. The rank of the access in the trace is irrelevant for finding cycles; if a cycle exists via an access to a location ℓ then a cycle exists via any such access performed at any time by the same thread. For an access $a_t(\ell)$ ($a \in \{r, w\}$, $t \in \mathbb{T}$, $\ell \in \mathbb{L}$), the summarised information is stored in PS_t as (a, ℓ). In addition, a read may be forgotten once a write to the same location has occurred, since any access a conflicting with an access $r(\ell)$ also conflicts with an access $w(\ell)$, but not vice versa. We extend the definition of the vertex-labelling function L to provide the summaries: for each PS_t ($t \in \mathbb{T}$), $L(PS_t)$ contains the tuples representing all completed accesses of t. Updating a P-summary PS_t with a set of accesses X can be formalised using a special union operator $\uplus$ as follows: $PS_t \uplus X = \{(a, \ell) \mid (a, \ell) \in PS_t \vee a_t(\ell) \in X \wedge (a = r \implies (w, \ell) \notin PS_t \wedge w_t(\ell) \notin X)\}$.

Consider a trace π and the corresponding summarised conflict graph SG_π. In SG_π, there are vertices for each instruction that has appeared but has not yet completed, plus the PS_t for all $t \in \mathbb{T}$. In the following, we say an instruction is *alive* iff it has a vertex in the summarised conflict graph.

A conflict (orientation) edge is represented by drawing an edge from a vertex v to a vertex v' iff $L(v) \prec/A^\pi\ L(v')$ and their labels conflict. A P-edge is represented by an edge from a vertex v to another vertex v' whenever $L(v)\ P^\pi/A^\pi\ L(v')$. Furthermore, there is a P-edge from each PS_t to every instruction vertex belonging to t.

To make it explicit how the summary graph is maintained we inductively define the summarised conflict graph for detecting sequential consistency violations. Given an $e \in \mathbb{V} \cup \{\checkmark_i \mid i \in I^\pi\}$, let $in(e)$ denote the instruction i that e belongs to.

Definition 6.4.8 (Summarised Conflict Graph)**.** *The* summarised conflict graph *of trace* π *is a graph* $SG_\pi = (V_\pi, E_\pi, L_\pi)$*, where* (V_π, E_π) *is a graph with* $PS \subseteq V_\pi$*, and* L_π *is the vertex-labelling function. The vertices in* $V_\pi \setminus PS$ *represent all the alive instructions.*

The summarised conflict graph *over the empty trace is the graph* $SG_\varepsilon = (PS, \emptyset, L)$*, where* $L(PS_t) = \emptyset$ *for all* $t \in \mathbb{T}$*.*

Consider a trace element $e \in \mathbb{V} \cup \{\checkmark_i \mid i \in I^\pi\}$ *of some thread* t*. We now build the graph* $SG_{\pi \prec e}$*.*

Let $\mathit{ExistsPAfter}(i) = (\exists i' \in V_\pi.\ L_\pi(i)\ P^\pi/A^\pi\ L_\pi(i')) \vee (L_\pi(i)\ P^\pi/A^\pi\ [e]^{\pi \prec e})$ *be the proposition stating that there exists in* SG_π*, or will exist in* $SG_{\pi \prec e}$*, a vertex that is* P*-after a given instruction* i*. Furthermore, let* $\mathit{AllPBeforeCompleted}(i) = \forall i' \in I.\ [i']\ P/A\ [i] \implies (i' \in V_\pi \wedge \checkmark_{i'} \in L_\pi(i')) \vee (e = \checkmark_{i'})$ *be the predicate stating that all instructions* P*-before some instruction* i *have been completed in* SG_π *or will be completed in* $SG_{\pi \prec e}$*. Finally, let* $\mathit{Del} = \{i \in V_\pi \mid (\checkmark_i \in L_\pi(i) \vee e = \checkmark_i) \wedge \mathit{ExistsPAfter}(i) \wedge \mathit{AllPBeforeCompleted}(i)\}$ *be the set of vertices that are to be deleted: for each such vertex* v*, there is at least one other vertex that is* P*-after* v*, and vertices* v' *that are* P*-before* v *have completed as well.*

The summarised conflict graph *of trace* $\pi \prec e$ *is defined by* $SG_{\pi \prec e} = (V_{\pi \prec e}, E_{\pi \prec e}, L_{\pi \prec e})$*, with*

- $V_{\pi \prec e} = V_\pi \cup \{in(e)\} \setminus \mathit{Del}$*;*
- $L_{\pi \prec e}(in(e)) = L_\pi(in(e)) \cup \{e\}$ *and* $L_{\pi \prec e}(PS_t) = L_\pi(PS_t) \uplus \bigcup_{i \in \mathit{Del}} [i]^{\pi \prec e}$*;*
- $E_{\pi \prec e} = E_\pi \setminus \mathit{DE} \cup \mathit{CA} \cup \mathit{CI} \cup \mathit{CT} \cup \mathit{PA} \cup \mathit{PT}$*, where*
  - $\mathit{DE} = \{(i, i') \mid i \in \mathit{Del} \vee i' \in \mathit{Del}\}$*, i.e., the set of edges between instructions to be deleted;*
  - $\mathit{CA} = \{(i, in(e)) \mid i \in V_{\pi \prec e} \wedge \exists e' \in L_{\pi \prec e}(i).\ e\ C^\pi/A^\pi\ e'\}$*, i.e., new* C*-edges between alive instructions and the instruction of* e*;*
  - $\mathit{CI} = \{(PS_{t'}, in(e)) \mid t' \neq t \wedge PS_{t'} \in V_{\pi \prec e} \wedge \exists e' \in L_{\pi \prec e}(PS_t).\ e\ C^\pi\ e'\}$*, i.e., new* C*-edges between* P*-summaries and the instruction of* e*;*
  - $\mathit{CT} = \{(PS_t, v) \mid v \in V_{\pi \prec e} \wedge (i', v) \in E_\pi \wedge i' \in \mathit{Del}\} \cup \{(v, PS_t) \mid v \in V_{\pi \prec e} \wedge (v, i') \in E_\pi \wedge i' \in \mathit{Del}\}$*, i.e., the* C*-edges to be connected to* PS_t *due to the deletion of instructions of* t*;*
  - $\mathit{PA} = \{(in(e), i) \mid i \in V_{\pi \prec e} \wedge \{e\}\ P^\pi/A^\pi\ L_{\pi \prec e}(i)\} \cup \{(i, in(e)) \mid i \in V_{\pi \prec e} \wedge L_{\pi \prec e}(i)\ P^\pi/A^\pi\ \{e\}\}$*, i.e., new* P*-edges between the instruction of* e *and alive instructions, and*
  - $\mathit{PT} = \{(PS_t, in(e))\}$*, i.e., the* P*-edge between* PS_t *and the instruction of* e*.*

The summarised conflict graph over an empty trace only contains P-summaries that are labelled with the empty set. When a new trace element $e \in \mathbb{V} \cup \{\checkmark_i \mid i \in I^\pi\}$ of some thread t is added to a trace π, forming $\pi \prec e$, SG_π is updated to $SG_{\pi \prec e}$.

The vertices V_π are updated by adding a vertex for the instruction of e if it is not already there. Furthermore, vertices that may be summarised (i.e., those in the set *Del*) are deleted. A vertex i is in *Del* iff i is a completed instruction (i.e., $\checkmark_i \in L_{\pi \prec e}(i)$), i is not the last instruction in the program order P^π (i.e., there is another vertex P-after i), and all vertices that P-precede i are completed as well. The summary (thus, deletion) of an instruction i removes the P^π-edge from preceding vertices, i.e., the summary of a vertex places the vertex's accesses in the corresponding P-summary; this signifies that those accesses P-precede those of alive vertices of the same thread. Hence, if a vertex i is summarised that has an alive P-preceding vertex i', then suddenly it appears as if i occurs *P-before* i' (instead of after). For similar reasons the P_t-last vertex in the trace may not be deleted. To maintain the correct program order relation, vertices that have uncompleted P-preceding vertices are thus not in *Del*.

The labelling function L_π is updated to $L_{\pi \prec e}$ for the instruction i of e by adding e to the set of accesses of vertex i. The labelling function is updated for PS_t by adding tuples

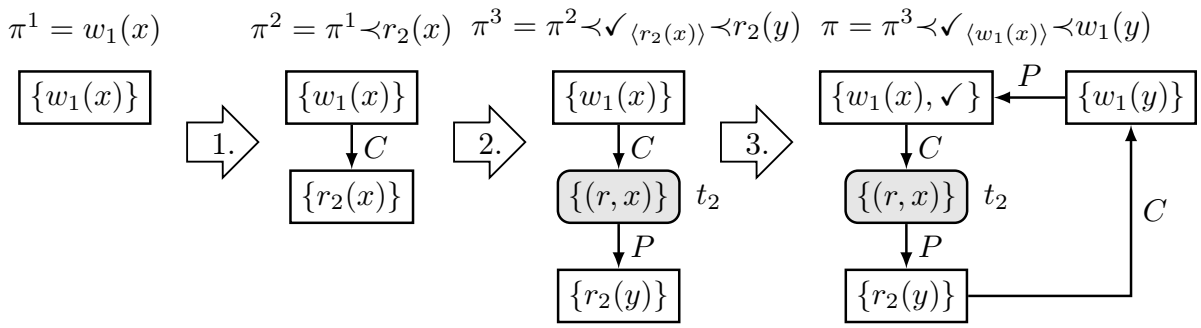


Figure 6.5: Construction of the summarised conflict graph for a non-SC trace $\pi = w_1(x) \prec r_2(x) \prec \checkmark_{\langle r_2(x)\rangle} \prec r_2(y) \prec \checkmark_{\langle w_1(x)\rangle} \prec w_1(y)$ with program $t_1 = \langle w_1(y)\rangle\ P_1\ \langle w_1(x)\rangle$ and $t_2 = \langle r_2(x)\rangle\ P_2\ \langle r_2(y)\rangle$ according to Definition 6.4.8.

for accesses of deleted vertices and removing from the resulting set all read access tuples for which a corresponding write access tuple is also in the set.

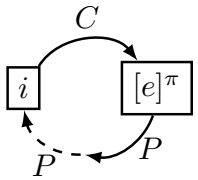
The edges E_π are updated to $E_{\pi \prec e}$ by removing edges to deleted vertices (in DE). If e is an access, then C-edges are drawn to the instruction vertex of e from alive vertices with conflicting accesses (the CA set of edges) and from P-summaries containing tuples representing conflicting accesses (the CI set of edges). Furthermore, any C-edges in SG_π connected to a vertex v of t to be deleted are redirected to PS_t (CT). Finally, the set of P-edges is updated by adding edges between alive vertices and the instruction of e (PA) and from PS_t and the instruction of e (PT). PT updates the transitive closure of P^π.

Example 6.4.9 (Example summarised conflict graph)**.** *Reconsider the program and non-SC trace in Example 6.4.7. Figure 6.5 shows how the summarised conflict graph for π is constructed according to Definition 6.4.8. P-summaries are represented by a* grey *box with rounded corners, alive instruction vertices are represented by a white box with sharp corners. To simplify the visualisation a P-summary is only shown when it has an edge to an alive instruction vertex. We omit indices of the $\checkmark$ as they can be derived from the corresponding vertex. The summarised conflict graph is constructed as follows as we traverse π:*

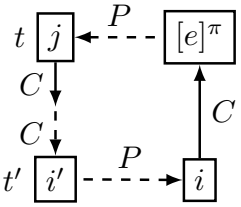
1. *The same as in Example 6.4.7: a vertex for $\langle r_2(x)\rangle$ is added and an edge is drawn for the conflict.*

2. *The completion of $\langle r_2(x)\rangle$ is marked by $\checkmark_{\langle r_2(x)\rangle}$. However, the vertex is not yet deleted as at that moment it is the last instruction of thread t_2. Next, when $r_2(y)$ arrives a new vertex is created for it. Since $r_2(y)$ follows $r_2(x)$ in the program order, the vertex labelled $\{r_2(x), \checkmark\}$ may be summarised: 1) the C-edge pointing to the vertex labelled $\{r_2(x), \checkmark\}$ is redirected to PS_{t_2}, 2) (r, x) is added to PS_{t_2}, and 3) a P-edge is drawn from PS_{t_2} to new vertex $\langle r_2(y)\rangle$ to represent the program order.*

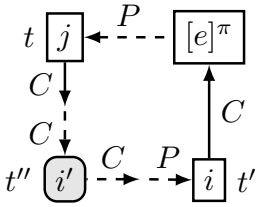
3. *First the completion of $\langle w_1(x)\rangle$ is marked. Although this instruction is completed, its vertex is kept alive (thus not summarised) because it is the last instruction of thread t_1. When access $w_1(y)$ arrives a new vertex is created for its corresponding instruction. The vertex labelled $\{w_1(x), \checkmark\}$ is still kept alive, removing and summarising the vertex now would be* unsound *since otherwise the direction of the program order would be represented in reverse order ($w_1(x)$ P $w_1(y)$ instead of $w_1(y)$ P $w_1(x)$). Finally, a cycle is found indicating that the trace is non-SC.*



(a) A cycle in $IG_{\pi \prec e}$ or $SG_{\pi \prec e}$ within the same thread program.



(b) A cycle in $IG_{\pi \prec e}$ between multiple thread programs.



(c) A cycle in $SG_{\pi \prec e}$ between that goes via a P-summary.

Figure 6.6: Possible cycles in $G_{\pi \prec e}$ if G_π does not contain cycles for some newly added $e \in \mathbb{V} \cup \{\checkmark_i \mid i \in I^\pi\}$ of a thread t.

Summarised conflict graph SG_π correctly represents all cycles in conflict graph IG_π.

Proposition 6.4.10. *Let π be a trace, then there is a cycle in IG_π iff there is a cycle in $SG_{\pi'}$ for some π' that is a prefix of π.*

Proof. We prove this by induction on π. For the smallest trace $\pi = \varepsilon$ the proposition trivially holds since there are no cycles in SG_ε.

Assume, for an Induction Hypothesis (IH), that the proposition holds for a trace π. We show that the proposition still holds for a trace $\pi \prec e$ with $e \in \mathbb{V} \cup \{\checkmark_i \mid i \in I^\pi\}$. By IH, it follows that there is a cycle in $G_{\pi \prec e}$ iff there is a cycle in $SG_{\pi'}$ for any prefix of π. If there is a cycle in IG_π or in SG_π then the proposition again holds by IH. It remains to be shown that if there is no cycle in IG_π and SG_π, then there is a cycle in $IG_{\pi \prec e}$ iff there is a cycle in $SG_{\pi \prec e}$.

Let t be the thread that executes e. Suppose that there are no cycles in IG_π and SG_π.

- $\Rightarrow$: There is a cycle in $IG_{\pi \prec e}$ iff adding e to IG_π introduces a cycle. Hence, e introduces a new C-edge to some instruction node $i \in IG_\pi$ of a thread t' oriented from i to $in(e)$. We distinguish two cases:

  - $t = t'$. In order to have a cycle, there must be a P-edge from $in(e)$ to i in $IG_{\pi \prec e}$, more specifically $\{e\}\ P^{\pi \prec e}/A^{\pi \prec e}\ L_{\pi \prec e}(i)$ as illustrated in Figure 6.6a. In this case, i could not have been deleted in $SG_{\pi \prec e}$ as e must be an access and *Del* does not contain vertices that have uncompleted P-preceding vertices. In $SG_{\pi \prec e}$, the C-edge and P-edges are contained in CA and PA, respectively. Hence, the cycle is in $SG_{\pi \prec e}$.
  - $t \neq t'$. As illustrated in Figure 6.6b, there must be a possibly empty P-path from $in(e)$ to a vertex j, a path consisting of at least one C-edge to a vertex i' of thread t', and a possibly empty P-path from i' to i. Note that j cannot be a P-summary of t, because it P-succeeds $in(e)$, and therefore, is not deleted by *Del*.

    If the cycle is formed between $in(e)$ and i directly, then the cycle is visible through the edge added by CA.

    We now show that the cycle is visible if the cycle goes through vertices other than $in(e)$ and i. If there is a P-path from vertex $in(e)$ to j and $j \neq in(e)$, then this path is still there in $SG_{\pi \prec e}$ as the edges were added by PA, j is not a P-summary, and the vertices on the path were not deleted by *Del* since they P-succeed $in(e)$. Furthermore, if along the cycle there is a conflict

via a P summary, then these remain visible in $SG_{\pi \prec e}$ through CT or past applications of CT. Similarly, any conflicts between alive nodes along the path were maintained by CA. If the cycle goes through the P-summary of t', i.e., $i' = PS_{t'}$, then PT has created an edge from i' to i in the past via PT. Alternatively, if the cycle does not go through $PS_{t'}$, then there is a possibly empty P-path from i' to i through uncompleted vertices, because any uncompleted vertex P-before i is not deleted by Del and the edges representing P were added by PA at some point. As all parts of the cycle are still represented in $SG_{\pi \prec e}$ in the same direction as in $IG_{\pi \prec e}$, the cycle is reported by $SG_{\pi \prec e}$ as well.

- $\Leftarrow$: There is a cycle in $SG_{\pi \prec e}$ iff e is an element of a cycle consisting of the accesses in π. Hence, e introduces a new conflict with some alive vertex $i \in SG_\pi$ of a thread t' oriented from i to $in(e)$. As PS_t has an edge towards $in(e)$ as well, the cycle cannot involve the PS_t. We distinguish two cases:
  - $t = t'$ or the cycle only goes through alive vertices. If $t = t'$, then the cycle cannot involve PS_t as the edge between it and $in(e)$ goes from PS_t to $in(e)$, while the conflict edge goes from i (which also succeeds PS_t) to $in(e)$. Hence, the cycle only goes through alive vertices as sketched by Figure 6.6a. As all nodes involved in the cycle are alive, the cycle must also be present in $IG_{\pi \prec e}$.
  - $t \neq t'$ and the cycle goes through at least one P-summary (Figure 6.6c). The C-edges in this cycle are represented in $IG_{\pi \prec e}$ as well: CA only adds C-edges between alive vertices, these are in $IG_{\pi \prec e}$ too; CI adds C-edges between alive vertices and P-summaries; and finally, CT adds C-edges from alive vertices to P-summaries, but only for vertices that are deleted (by DE), and direction is maintained. The P-edges in the cycle are represented in $IG_{\pi \prec e}$ as well: PA ads P-edges between alive vertices, these are also in in $IG_{\pi \prec e}$; and, any non-P-summary has an incoming P-edge from its P-summary, since $P_{t''}$ is total for any thread $t'' \in \mathbb{T}$ the edge must also be represented in $IG_{\pi \prec e}$. Furthermore, for edges to and from P-summaries there must be some vertex $IG_{\pi \prec e}$ to and from vertices represented by P-summaries or alive vertices in $SG_{\pi \prec e}$. It follows that also $IG_{\pi \prec e}$ contains a cycle. □

6.5 Deriving Locks and Delays with Model Checking

The algorithm of Shasha and Snir is a static analysis method. It depends on detecting conflicting accesses at compile time, which can be pessimistic [5]. With a monitor actual execution traces can be verified. However, it is inefficient to consider all possible reorderings and interleavings of a program.

In this section, we adapt the summarised conflict graph, in a number of iterations, to a monitor that can be used to efficiently report a sufficient delay set by means of a model checker. Cycles *within* instructions will be considered separately by a preprocessing step.

We first adapt the summarised conflict graph to reduce the required state space of the model checker. Then, we adapt the summarised conflict graph further to optimise cycle detection.

Reduce the state space. The following changes are applied to the summarised conflict graph for state space reduction:

1. *The orientation of conflicts in the trace is omitted.* For the derivation of the delay relation the orientation of conflicts is not relevant: all the required information is in $P \cup C$.

2. *Traces are expected in program order.* As is evident from the theory of Sasha and Snir it is sufficient to consider only the program order and the conflict relation. The model checker will explore all possible SC-traces, and not all SC-serialisable traces.

3. *Entire instructions are added to the trace at once* instead of one access at a time. Since traces are in program order, it is sufficient to consider entire instructions at once for the detection of cycles in P/A. Drawing edges according to the theory of Sasha and Snir covers all possible interleavings of individual accesses.

With the above changes, a model checker can explore all possible SC-traces, and each time when encountering a transition associated with an instruction j of a thread t, consider j's predecessor j' in the P-order of t to have terminated. Therefore, it will summarise vertex j' in the summarised conflict graph and add a vertex j.

Optimisation of cycle detection. For efficiency we want to be able to detect cycles by comparing as few vertices as possible. To achieve this we make two changes to the summarised conflict graph which summarises conflicts (and their direction via P-edges). Cycle appear as self-loops of these summarised conflicts.

The *P-summaries are replaced by a function*; the content of a PS_t is stored via a function $PS(t)$ that returns the summarised access tuples for t. Any edges that are lost due to the removal of PS_t can be derived from the PS function and alive vertices.

Furthermore, *the C-relation is summarised.* More precisely, the transitive closure of the C-relation in combination with the P-relation stored for each alive instruction vertex. For each alive vertices the accesses of the corresponding thread are stored such that delays can be derived from the C-summary.

The adapted summary graph for a trace π is denoted by AG_π.

Summary of the C-Relation. Consider a trace π and an alive instruction $i \in V_\pi$. We refer with $th(i)$ to the thread executing instruction i. Given an alive instruction $k \in V_\pi$ the set of accesses of i conflicting with k is defined as $C_i(k) = \{a \in [i] \mid a' \in [k] \wedge a\ C\ a'\}$.

The transitive in- and out-C-summary of an instruction i, denoted by $CS^{in}_\pi(i)$ and $CS^{out}_\pi(i)$, respectively, consists of pairs of threads and accesses. Given a thread t and a set of accesses X, the presence of such a pair (t, X) in $CS^{in}_\pi(i)$ indicates that thread t conflicts either directly or indirectly with instruction i, and X is the set of accesses of $th(i)$ that are involved in the conflict.

The $CS^{in}_\pi(i)$ is inductively defined as follows ($CS^{out}_\pi(i)$ is defined symmetrically). Initially, $CS^{in}_\varepsilon(i) = \emptyset$.

Consider an observed instruction j and its P-preceding instruction j' (which will be summarised after computing the C-summaries). The cases for the inductive step $CS^{in}_{\pi \prec j}(i)$ are illustrated in Figure 6.7. The traversed instruction i is represented by the top right vertex. The dashed diagonal edge represents a C-summary edge and is the consequence of the vertical and horizontal edges. These C-summary edges are collected in the set $CS_{\pi \prec i}$ which consists of the sets *C-P*, *CS-P*, *C-CS*, *C-C*, and *CS-CS*. A C-summary edge from vertex k to i with sets of access X and Y at its tail and head, respectively, indicates that the thread of k is in the incoming C-summary of i and Y is the set of accesses of the tread of i that are involved in the conflict. Similarly, the edge indicates that the thread

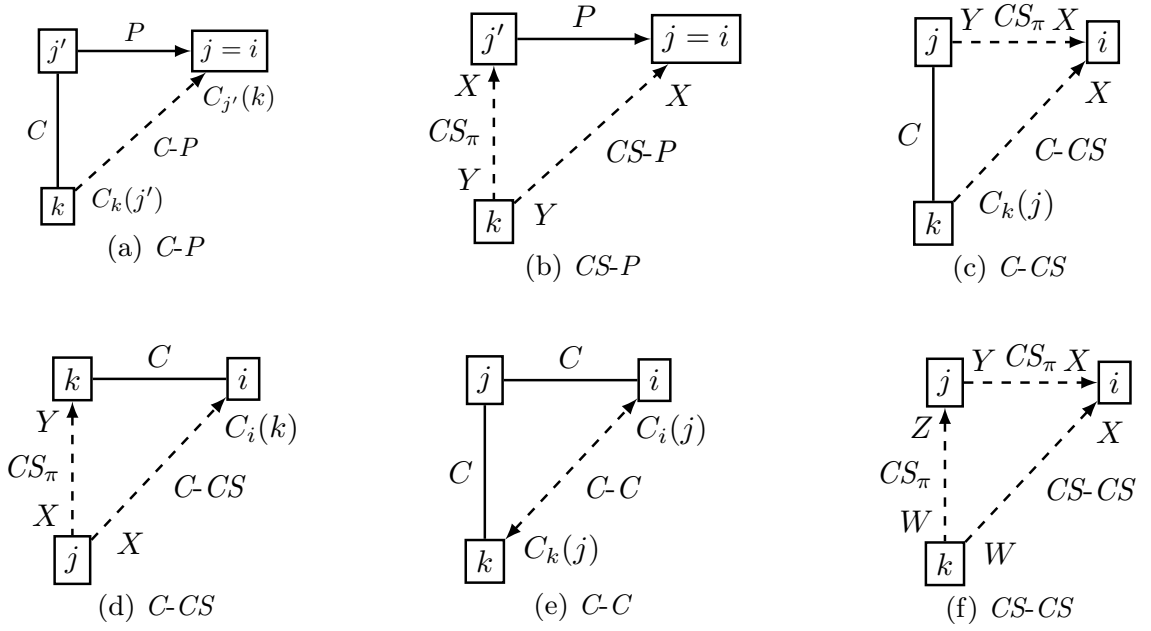


Figure 6.7: C-summaries for some trace π, vertices $i, i' \in V_{\pi \prec j}$, and traversed vertex $j \in V_\pi \setminus V_{\pi \prec j}$; the diagonal edge is the consequence of the vertical and horizontal edges.

of i is in the outgoing C-summary of k and that X is the set of accesses of the thread of k that are involved in the conflict. Formally, $CS^{in}_{\pi \prec j}(i)$ is defined as follows:

$$CS^{in}_{\pi \prec j}(i) = CS^{in}_{\pi}(i) \setminus \{th(j)\} \cup C\text{-}P^{in} \cup CS\text{-}P^{in} \cup C\text{-}CS^{in} \cup CS\text{-}C^{in} \cup C\text{-}C \cup CS\text{-}CS^{in}$$

where

- $C\text{-}P^{in} = \{(th(k), C_{j'}(k)) \mid i = j \wedge k \in V_\pi \setminus \{i\} \wedge L_\pi(k)\ C^\pi\ L_\pi(j')\}$ is the set of thread/access pairs directly conflicting with the P-predecessor of j (Figure 6.7a);
- $CS\text{-}P^{in} = \{(th(k), X) \mid i = j \wedge k \in V_\pi \setminus \{i\} \wedge (th(i'), X) \in CS^{in}_\pi(th(i))\}$ is the set of thread/access pairs indirectly conflicting with the P-predecessor of j (Figure 6.7b);
- $C\text{-}CS^{in} = \{(th(k), X) \mid k \in V_\pi \wedge j\ C^\pi\ k \wedge (th(j), X) \in CS^{in}_\pi(i)\}$ is the set of thread/access pairs that conflict indirectly with i via a direct conflict and a C-summary (Figure 6.7c);
- $CS\text{-}C^{in} = \{(th(j), C_i(k)) \mid k \in V_\pi \wedge k\ C^\pi\ i \wedge (th(j), X) \in CS^{in}_\pi(k)\}$ is the set of thread/access pairs that conflict indirectly with i via a C-summary and a direct conflict (Figure 6.7d);
- $C\text{-}C = \{(th(k), C_i(j)) \mid k \in V_\pi \wedge k\ C^\pi\ j\ C^\pi\ i\}$ is the set of thread/access pairs that conflict with i indirectly via two C-edges (Figure 6.7e); and
- $CS\text{-}CS^{in} = \{(t, X) \mid (th(j), X) \in CS_\pi(i) \wedge (t, Y) \in CS_\pi(j)\}$ is the transitive step merging two incoming C-summaries. (Figure 6.7f).

Moreover, $CS^{in}_{\pi \prec j}(j') = \emptyset$, and $CS^{out}_{\pi \prec j}(k)$ is defined symmetrically.

In the definition of $CS^{in}_{\pi \prec j}(i)$, *first*, the old set is recalled through $CS^{in}_\pi(i)$. *Second*, the thread of j is discarded. As j' has a P-edge to j any vertices k with incoming

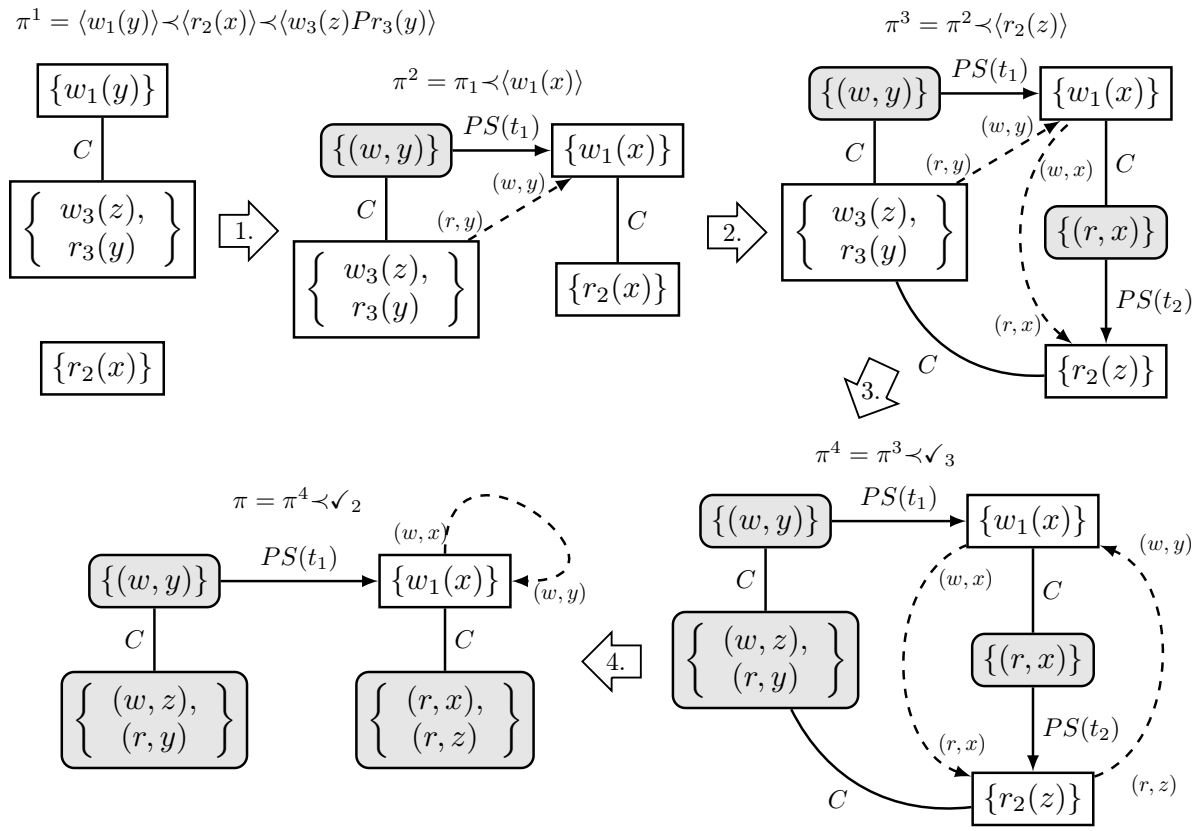


Figure 6.8: Construction of the adapted summarised conflict graph for a trace of instructions $\pi = \langle w_1(y)\rangle \prec \langle r_2(x)\rangle \prec \langle w_3(z)\ P_3\ r_3(y)\rangle \prec \langle r_2(z)\rangle$ for a program $t_1 = \langle w_1(y)\rangle\ P_1\ \langle w_1(x)\rangle$, $t_2 = \langle r_2(x)\rangle\ P_2\ \langle r_2(z)\rangle$, and $t_3 = \langle w_3(z)\ P_3\ r_3(y)\rangle$; the C-summaries are indicated by dashed arrows.

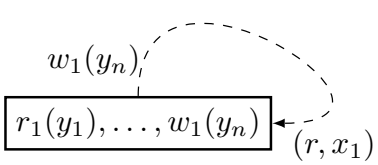
conflicts from j' will not be able to form a cycle together with j. *Finally*, the conflicts are summarised through the sets $C\text{-}P^{in}_{\pi\prec j}$, $C\text{-}CS^{in}$, $C\text{-}C$, and $CS\text{-}CS^{in}$.

Example 6.5.1 demonstrates how the conflict relation is summarised to a self-loop.

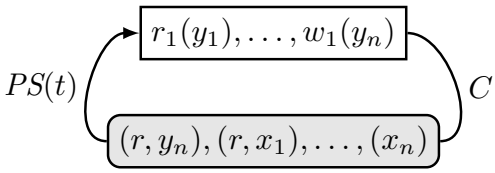
Example 6.5.1 (Detecting a cycle via C-summaries)**.** *Consider the program* $t_1 = \langle w_1(y)\rangle\ P_1\ \langle w_1(x)\rangle$, $t_2 = \langle r_2(x)\rangle\ P_2\ \langle r_2(z)\rangle$, *and* $t_3 = \langle w_3(z)\rangle\ P_2\ \langle r_3(y)\rangle$. *The program may execute the non-SC trace* $\pi' = w_1(x) \prec \checkmark_{\langle w_1(x)\rangle} \prec r_2(x) \prec \checkmark_{\langle r_2(x)\rangle} \prec r_2(z) \prec \checkmark_{\langle r_1(z)\rangle} \prec w_3(z) \prec r_3(y) \prec w_1(y)$. *The adapted summarised conflict graph detects this SC violation in the trace of instructions* $\pi = \langle w_1(y)\rangle \prec \langle r_2(x)\rangle \prec \langle w_3(z)\ P_3\ r_3(y)\rangle \prec \langle r_2(z)\rangle \prec \checkmark_3 \prec \checkmark_2$.

Figure 6.8 illustrates the construction of the adapted summarised conflict graph AG_π *for trace* π. *The* P*-summary function* $PS(t)$ *for some thread* t *is depicted as a grey vertex (though it is a function, not a vertex) with rounded corners that has a* $PS(t)$*-edge to the vertices of alive instructions of* t. *The* C*-summaries are indicated by dashed arrows. To simplify the illustration, the* C*-summaries are omitted for threads that do not have alive instructions. Below we discuss how the adapted summary graph is constructed and show that, using the* C*-summaries, the (summarised) cycle of* π' *can eventually be found as a self-loop.*

After transitioning the first instruction of each thread we obtain the conflict graph π^1 *shown in the top left corner. Construction of the adapted summarised conflict graph proceeds as follows:*



(a) Self-loop indicating "(r, x_1) before $w_1(y_n)$" from the C-summary.



(b) Self-loop indicating "(r, y_n) before $w_1(y_n)$" from the P-summary (grey node represents value of function $PS(t)$).

Figure 6.9: Deriving delays from self-loops.

1. *When instruction $\langle w_1(x)\rangle$ is traversed by the model checker the conflict from $r_3(y)$ to $w_1(x)$ via $w_1(y)$ is stored by the C-summaries: $CS^{in}_{\pi^2}(\langle w_1(x)\rangle) = \{t_3\}$ via C-$P^{in}_{\pi^2}$, and $CS^{out}_{\pi^2}(\langle w_3(z)\ P_3\ r_3(y)\rangle) = \{t_1\}$ via C-$P^{out}_{\pi^2}$ (Figure 6.7a). Furthermore, instruction $\langle w_1(y)\rangle$ is summarised and stored by adding (w, y) to $PS(t_1)$*

2. *Next, instruction $\langle r_2(z)\rangle$ is traversed. An edge representing the conflict between $w_3(z)$ and $r_2(z)$ is added. The conflict between $\langle w_1(x)\rangle$ and the instruction $\langle r_2(x)\rangle$ that will be summarised is stored in $CS^{out}_{\pi^3}(\langle w_1(x)\rangle)$ and $CS^{in}_{\pi^3}(\langle r_2(z)\rangle)$ via C-$P^{in}_{\pi^3}$ and C-$P^{out}_{\pi^3}$, respectively (Figure 6.7a). Finally, the $\langle r_2(x)\rangle$ instruction is summarised by adding (r, x) to $PS(t_2)$ and the conflict between $w_3(z)$ and $r_2(z)$ is now represented via $PS(t_2)$.*

   When the dashed arrows are omitted the resulting conflict graph corresponds to the summarised graph SG_{π^3} for the non-SC trace π'. The cycle is now visible via PS- and C-edges.

3. *Thread t_3 terminates. The conflict path from $\langle r_2(z)\rangle$ to $\langle w_1(x)\rangle$ via vertex $\langle w_3(z)\ P_3\ r_3(y)\rangle$ and its C-summary to $\langle w_1(x)\rangle$ is added to the C-summaries via C-CS: $CS^{in}_{\pi^4}(\langle w_1(x)\rangle) = \{t_2, t_3\}$, and $CS^{out}_{\pi^4}(\langle r_2(z)\rangle) = \{t_1\}$ (Figure 6.7c). Finally, vertex $\langle w_3(z)Pr_3(y)\rangle$ is summarised to $PS(t_3) = \{(w, z), (r, y)\}$ and any conflicts on $\langle w_3(z)Pr_3(y)\rangle$ are updated to target $PS(t_3)$.*

4. *Finally, thread t_2 terminates. The subset CS-CS^{in} in $CS^{in}_{\pi^4 \prec \langle r_2(z)\rangle}$ summarises the two C-summary edges in the graph AG_{π^4} (Figure 6.7f). Since $t_2 \in CS^{in}_{\pi^4}(\langle w_1(x)\rangle)$, we now have a C-summary self-loop on vertex $\langle w_1(x)\rangle$ via $t_1 \in CS^{in}_{\pi^4 \prec \langle r_2(z)\rangle}(\langle w_1(x)\rangle)$ formally summarising the cycle first visible in the graph AG_{π^3}. The loop shows that $w_1(x)$ should be delayed until writes to y have been completed. As the last step, the final instruction of thread t_2 is summarised.*

All cycles in $C/A \cup P/A$ eventually appear as self-loops if all SC-traces are considered by the model checker. To derive delays two kinds of self-loops must be considered as illustrated by Figure 6.9. Cycles involving instructions of multiple threads can be detected through C-summary self-loops (Figure 6.9a and Example 6.5.1). A self-loop on an instruction i such as the one presented in Figure 6.9a indicates that the accesses of $CS^{out}_{\pi}(i)$ must be delayed until those of $CS^{in}_{\pi}(i)$ are completed. To detect cycles between instructions of the same thread the P-summary function must be considered (Figure 6.9b); similar to the summary graph, a cycle between an instruction i that is P-after another instruction of the same thread appears as a conflict of an instruction i on its P-summary function. A self-loop on an instruction i via its P-summary such as the one presented in Figure 6.9 indicates that accesses of the instruction must be delayed until conflicting accesses of the P-summary have been completed.

Reconsider Example 6.5.1. If all SC-traces are considered, then a self-loop appears on the final instruction of each thread. The scenario presented gives rise to a self-loop from which the delay (w, y) before $w_1(x)$" is derived. Furthermore, when $\checkmark_1$ is traversed in step 4 instead of $\checkmark_3$ the delay "(r, x) before $r_2(z)$" is derived. Similarly, when $\checkmark_1$ and $\checkmark_2$ are traversed in step 3 and 4, then the delay "$w_3(z)$ before $r_3(y)$" is derived.

Focussing the Model-Checker. It is possible to initially over-approximate the potential for sequential consistency violations in a static analysis algorithm. Such an analysis can identify the instructions that are *safe*, i.e., of which the involved accesses cannot be separated by conflicting accesses of other threads. This helps to subsequently focus the model checking algorithm on unsafe instructions, i.e., when traversing a transition associated with a safe instruction i, $AG_{\pi \prec i}$ is set to AG_π and no cycle detection needs to take place. For static analysis one can use any method that identifies at least all critical delays in the program specification (e.g., the algorithm of Shasha and Snir). However, as their algorithm involves iterating over all cycles in the dependency graph containing a node for every instruction in the program, we have opted for a less precise, but faster alternative: first, we construct a dependency graph involving all instructions, and represent both the P- and C-relations as edges between the instruction-nodes. Then, we identify all strongly connected components (using Tarjan's algorithm [199]) and subsequently identify the components that involve at least one P-edge and one C-edge. In addition, if either at least two threads are involved with multiple instructions, i.e., instructions that are in the component, or the involved C-edge represents conflicts between more than two accesses, then the component includes cycles that may constitute violations. In that case, all instructions in that component should be marked unsafe.

To further reduce the computational effort, state space reduction techniques such as *Partial Order Reduction* [89, 164, 202] can be applied. Prominent approaches to POR are the *Ample set approach* [164], the *Stubborn set approach* [202], and the *Sleep set approach* [89]. In each approach, a subset of transitions is identified, either statically or dynamically while exploring the state space, that is sufficient to explore all behaviour relevant for the property to be verified. In the context of SC violation detection, we use the Ample set approach; identifying the set $\mathcal{A}$ consisting of all instructions that do not access shared variables. Pruning can then be performed as follows: from any state in which at least one thread t can only perform instructions in $\mathcal{A}$, only those transitions of t can be traversed, ignoring the others, as they will also be enabled in the successor states. However, special attention is needed to ensure that in cycles of such instructions, at least one state is fully explored; if not, the exploration may be restricted too much, missing any behaviour beyond such cycles. Note that in this approach, $\mathcal{A}$ is trivially a subset of the set of safe instructions as defined above. For any version of POR to be compatible with SC violation detection, this is a requirement.

Support for conditionally enabled instructions. So far, we have always considered instructions that are *not guarded*, i.e., that are not conditionally enabled depending on whether the system state satisfies some Boolean predicate. However, in (multi-threaded) programs, the use of guards is very common, and in fact makes its necessary to analyse the dynamic semantics of a program, as opposed to statically analysing it. Without taking precautions, our algorithm may miss violations in which read accesses associated to checking a guard are involved, in particular when the guard evaluates to **false**. Namely, when in a particular system state, an instruction is disabled, it is by definition not observable as an outgoing transition from the associated state in the state space, even

Table 6.1: Runtime results (secs.) of standard exploration and SC violation checking

Test case	$\lvert\mathbb{T}\rvert$	$\lvert\boldsymbol{I}\rvert$	**Original**		**Monitored**		**SA**	**SC-check time**			$\lvert\boldsymbol{D}\rvert$
			#States	**#Trans.**	**#States**	**#Trans.**		**Full**	**POR**	**OTF**	
adding.1	2	6	7.4k	11.1k	722.4k	1.4m	3/3	2.2k	2.2k	249.2	22
adding.2	2	6	836.8k	1.3m	92.4m	169.2m	3/3	241.2k	239.5k	3.2k	22
anderson.1	2	12	352.6k	704.3k	t.o.	t.o.	3/3	t.o.	t.o.	93.2	t.o.
anderson.2	3	18	1.5k	3.7k	t.o.	t.o.	4/4	t.o.	t.o.	55.7k	t.o.
backery.1	2	16	1.5k	2.7k	t.o.	t.o.	4/4	t.o.	t.o.	10.1	t.o.
mcs.2	3	39	1.4k	3.2k	2.9m	8.5m	7/7	13.4k	12.3k	6.2k	143
rushhour.2	8	17	2.3k	12.6k	n.a.	n.a.	0/36	n.a.	n.a.	n.a.	n.a.
telephony.1	2	62	501	1.4k	31.5m	63.1m	8/8	251.5k	249.4k	80.6k	4.5k

though in the corresponding program, read accesses would need to be performed to determine that the instruction is indeed not enabled.

Our algorithm can be made sound by making the read accesses for guard checking observable. To do so, each time a transition of a thread t is explored, we add its corresponding instruction to the dependency graph, but also a second vertex representing all the reads needed in the source state to evaluate the guards of all the instruction options (enabled or not) in the corresponding state of the model. Before doing so, similarly, the existing vertex of t for guard checking in the graph is summarised, together with the current instruction vertex of t. Finally, to handle situations correctly in which a thread t cannot execute any instruction because all its options are disabled, a 'dummy' transition for t can be explored in the state space, by which the read accesses needed to check the guards of t can still be made visible in the graph, and summarised once a follow-up (possibly also dummy) transition of t is explored.

6.6 Implementation

In the literature little attention is given to preservation of SC-serialisability for DSLs when generating code. To this end, we chose to implement the presented model checking approach for the Slco DSL. In Slco, multi-threaded programs can be modelled using state machines and shared variables of types Boolean, Integer, Byte, and arrays containing values of those types. For the details, we refer the reader to [71]. We have implemented a tool in Python using the packages textX [65] and Jinja2[1] that statically under-approximates whether an Slco model can be transformed to code without the use of synchronisation mechanisms. If possible sequential consistency violations exist, the tool produces an mCRL2 model capturing the semantics of the Slco model and our SC violation resolution algorithm, so that the mCRL2 toolset [54] can be used for more precise analysis via state space exploration. This analysis is focussed on those variables that have been identified by the static analysis as possibly critical in sequential consistency violations.

6.7 Experimental Results

We have conducted numerous experiments on the DAS-5 cluster [17], with nodes equipped with an Intel Haswell E5-2630-v3 2.4 GHz CPU, 64 GB memory, and running CentOS Linux 7.4.

[1] http://jinja.pocoo.org.

Table 6.1 contains a representative selection of the obtained results.[2] All models have been taken from the BEEM benchmark set [162]. The cases are all relatively small, yet the BEEM models represent real protocols and systems.

For each test case, the number of threads ($|\mathbb{T}|$) and distinct instructions $|I|$ are listed. If values exceed a thousand, then they reported in thousands (k), or even millions (m). The time-out for exploration was set to 80 hours; 't.o.' indicates that the analysis timed-out. The state space before and after application of the monitor are shown in the *Original* and *Monitored* columns, where *#States* indicates the number states and *#Trans.* indicates the number of transitions.

The cases demonstrate that the state space explosion, when keeping track of P- and C-summaries is considerable, compared to the original state spaces. This is a big challenge, currently making the applicability of the model checking approach very limited. It is also to be expected, since it involves bookkeeping of dependencies between instructions, including the accesses that cause those dependencies.

The *SA* column indicates the outcome of the static analysis, in the form ⟨number of variables requiring analysis⟩ / ⟨total number of variables in the system⟩. In all cases, the runtime of static analysis is negligible. If statis analysis was able to determine that no violations can occur, then there was no need to conduct state space exploration. In this case runtime and monitored state space are reported as not applicable (n.a.).

The *SC-check time* column shows the runtime of exhaustive analysis (*Full*), exhaustive analysis with POR (*POR*), and on-the-fly resolution (*OTF*). The output of the first two setups is a list of delays, the size of which is given in the $|D|$ column. In the on-the-fly resolution setup, detected cyclic delays will be directly marked for locking, after which the involved variables are no longer tracked in the remainder of the analysis, allowing for early termination if at any point, no more variables are tracked. The drawback is that the produced result is coarser: if there are accesses to a variable x involved in cyclic delays, then the outcome is that *all* accesses to x should be protected (for instance, all accesses should be included in a transaction when using transactional memory, or variable x should be protected by a lock).

In all cases, with exception of the *rushhour.2* case, on-the-fly resolution helps to reduce the analysis time significantly. In the *anderson.1*, *anderson.2*, *bakery.1*, and *phils.1* cases, on-the-fly resolution is able to produce a result while the others time out. Its effectiveness is hard to predict, though; notice that for *anderson.2*, many more states and transitions need to be explored than for *anderson.1*, even though the two cases are variants of the same model, and the original state space of the former is much smaller than the one of the latter. POR has little effect. Our applied POR (Ample set) is quite conservative, and in the future, we will conduct experiments with more effective variants of POR.

We have not experimentally compared our approach to others, due to the fact that other model checking-based algorithms do not provide the same features, or reason about interleavings of accesses as opposed to instructions. Note that a model checking approach in which all interleavings of accesses are considered would scale much worse; if we need to consider n parallel instructions, each consisting of m accesses, then our approach explores worst-case 2^n states, while the other approach would have to explore $(m+1)^n$ states, and up to $n \cdot (m+1)^n$ and $n \cdot m \cdot (m+1)^n$ transitions when a complete thread-local access ordering and partial orderings are assumed, respectively. Nevertheless, for practical use, the scalability issues have to be overcome. We will work on this, and see multiple possibilities for improvement (see Section 6.8).

[2] Tools and models can be downloaded at `http://www.win.tue.nl/~awijs/seq_con`.

6.8 Conclusions

In this chapter we have proposed a monitor that detects sequential consistency violations. The correctness of the monitor is deduced from the theory of Shasha and Snir; thereby achieving support for a wide variety of weak memory models.

Furthermore, we showed how to safely summarise instructions once all the preceding instructions are completed. In the worst case, if accesses of transactions appear in reverse order the memory footprint is equivalent to that of a conflict graph that is not summarised. However, in most cases the summary vertices will get saturated and efficiently summarise most of the accesses.

The monitor is further developed for efficient use in model checkers. The monitor treats SC-traces of instructions, and therefore, instructions can be treated as a whole; reducing the number of interleaving accesses that need to be explored. In addition, we summarise the conflict relation between instructions such that cycles eventually appear as self-loops on the vertices of the conflict graph.

To further improve the efficiency of the conflict graphs we filter out accesses that are deemed safe by a static analysis: these safe accesses are ignored by the conflict graphs and, furthermore, allow the application of POR in the model checker.

Finally, we have employed the approach in the mCRL2 model checker on a number of Beem models expressed in the Slco DSL. The results indicate that in order to obtain an efficient model checking setup for sequential consistency violation checking, further improvements are required. We experimented with on-the-fly resolution of conflicts, which helps to reduce the runtimes, but a more sophisticated resolution procedure is required to not overestimate the amount of required synchronisation in the multi-threaded program.

Future Work Scalability is currently still a source of concern. However, we see potential to make the analysis more efficient. Techniques that iteratively search different parts of the state space, such as incremental counter-example construction [220], could be very effective. These kinds of techniques must be adapted to take sequential consistency resolution results into account as the analysis continues. Furthermore, parallelisation using graphics processors could be very effective, which has been demonstrated for model checking before [211, 212, 221].

More advanced POR techniques could reduce the state space even further than the Ample set approach employed in this work. It will have to be investigated how restrictive these approaches can be without missing any information essential for SC violation detection.

Finally, we would like to investigate the summary of conflict graphs based on the theory of Alglave and Maranget [6]. This would further extend the support for weak memory models by our model checking technique to memory models that allow *store atomicity relaxation* (e.g., the Power and ARM architectures).

Chapter 7

A Framework for Verified, Model-Driven Construction of Component Software

*We present the Simple Language of Communicating Objects (*Slco*) framework, which builds on results from our research on applying formal methods for correct and efficient model-driven development of multi-component software. At the core is a domain specific language called* Slco *that specifies software behaviour. In this chapter, we discuss the language, give an overview of the features of the framework, and discuss our roadmap for the future.*

New to the Slco *framework are three features that are the results of this thesis. First, a first step to model-to-model transformation verification is offered based on the transformation verification theory developed in Chapter 3. Second, due to the theory presented in Chapter 4, the divergence-preserving branching bisimulation equivalence relation may be compositionally applied during verification of* Slco *models. Third, the sequential consistency violation resolution algorithm described in Chapter 6 is applied to generate efficient Java implementations for* Slco *models.*

This chapter is an extension of

[63] de Putter, S., Wijs, A., and Zhang, D. The SLCO Framework for Verified, Model-Driven Construction of Component Software. In *FACS* (2018), LNCS, Springer, pp. 288–296

7.1 Introduction

The development of complex, multi-component software is time-consuming and error-prone. One important cause is that there are multiple concerns to address. In particular, the software should be functionally correct, but also efficient. Careless optimisation of code may introduce bugs and make it less obvious to reason about the core functionality. To improve this, it is crucial that techniques are developed that make every step in the development work flow systematic and transparent.

With the Simple Language of Communicating Objects (SLCO) framework [71, 224], we conduct research on the development of techniques for this purpose. Key characteristics are

1. The use of a Domain-Specific Language (DSL) based on well-known software engineering concepts, i.e., objects, variables, state machines, and sequences of instructions,
2. Formal verification in every development step, from model to code, that does not require expert verification knowledge from the developer, and
3. Optimised code generation, by which (parallel) programming challenges are hidden from the developer.

The framework uses the *model-driven software development* methodology, in which models are constructed and transformed to other models and code by means of *model transformations*. The framework makes use of a verified code generator [224, 225]. Furthermore, the framework supports some verification of model-transformations; extension to support the complete theory of Chapter 3 is planned in the near future.

Contributions This chapter presents a brief overview of the SLCO framework. The overview covers all recent work involving the new SLCO 2.0 language; formal verification of models, model visualisation, verification of model-to-model transformations, and verified code generation with efficient use of synchronisation constructs while at the same time guaranteeing sequential consistency up to *observation*. These observably Sequential Consistent (SC) implementations are indistinguishable from implementations that execute in the program order specified by the source SLCO model and where memory accesses are serviced from a single FIFO queue. Furthermore, special attention is given to recent developments in verifying SLCO model-to-model transformations and the efficient use of synchronisation constructs.

As a first step towards verified SLCO transformations the framework offers a translation from transformations defined in HENSHIN [12] to REFINER (transformation) rule systems. These rule systems can then be verified using REFINER, a tool implementing the transformation verification approach presented in Chapter 3. Currently, only transformations over user-defined actions can be translated to rule systems.

The SLCO framework offers a (semantics preserving) translation to SLCO-AL (SLCO Annotated Level), a slightly more detailed version of the language that allows the specification of *synchronisation fences*. The algorithm presented in Chapter 6 is used to factor out certain statements from atomic sequences of statements (reducing the length of the critical section) that require heavy-weight synchronisation constructs and to introduce light-weight fences where they are a safe alternative. The introduced synchronisation constructs restrict the executions of the SLCO-AL model to those that are SC with the source SLCO model.

Structure of the chapter In Section 7.2 related work is discussed. Next, we introduce the Slco 2.0 language and mention differences with respect to the previous version of Slco in Section 7.3. Section 7.4 describes the features offered by the framework. In particular, we elaborate on the verification of model-to-model transformations, code generation, and the translation to Slco-al. Finally, in Section 7.5 the chapter is concluded with a roadmap consisting of planned future work for the Slco framework.

7.2 Related Work

In most related work on model transformations, no verification is done (e.g. [66, 178]) or only on either model-to-model or model-to-code transformations [173]. Some techniques cover both, e.g. [152], but they do not address the *direct* verification of transformations. This means that correctness of a transformation cannot be determined once-and-for-all; instead, every time it is applied, its result has to be verified. Furthermore, a few transformation steps may quickly render verification infeasible [11].

The Slco framework has yet to achieve direct verification of *all* transformations, but transformations between transition structures consisting of actions can already be verified.

Scade 6 [69], Simulink [200], and Event-B [3] are frameworks offering features similar to Slco. All frameworks offer automatic code generation and verification methods for their models. Scade can make use of Lustre's verified compiler [31] to generated code. Both Simulink and Event-B support verification of generated code [78, 157], however, to our knowledge the generators have not been mechanically verified and, thus, require some form of consistency verification between model and code.

Unlike Scade, Slco is not limited to the sampling-actuating model of control engineering, but can specify such systems via user defined actions serving as sampling and actuating calls. Of these frameworks, only Event-B offers verification of refinement transformations. Slco, in addition, also supports verification of other kinds of model-transformations. Finally, similar to Scade, the Slco code generator is mechanically verified and preserves certain correctness criteria without the need for a consistency check.

7.3 An introduction to Slco 2.0 Language

The second version of Slco is the core of the framework. The Slco DSL should be used in the first development step to specify the intended functionality of a system. Slco has been designed to model systems consisting of concurrent, communicating components at a convenient level of abstraction. It has a formal semantics.[1] New to version 2.0 is the support for array, user defined actions, composite statements, the specification of a channel's buffer capacity, and transition priorities. We will introduce these additions together with the rest of Slco.

Slco models consist of a finite number of *classes*, which can be instantiated as *objects*, *channels* for communication between objects, and user defined *actions*; each are declared in their own section of the model.

A *class* consists of a finite number of concurrent *state machines*, and *ports* and *variables* shared by them that can be used for communication.

Variables are of type `Integer`, `Byte`, `Boolean`, or Array of one of these. Furthermore, state machines can have private variables that are only accessible by the owning

[1] See `http://www.win.tue.nl/~awijs/SLCO/SLCO2doc.pdf`.

state machine. Variables are declared in the `variables` section of classes or state machines: `Integer x := y + 1` declares an integer variable named `x` that initially has the value of the expression `y + 1`.

A *channel* connects two ports of two objects; it is used to send messages between state machines of two different objects. A channel accepts messages (optionally with parameters of types `Integer`, `Byte`, or `Boolean`), it is either synchronous or asynchronous. Furthermore, an asynchronous channels is either lossless or lossy (lossy means that it may lose messages at any time). In case a channel is asynchronous, a buffer size can be defined, which is by default 1. Let `p` and `q` be objects with ports `InOut`, `In`, and `Out`, then `c(Byte) sync between p.InOut and q.InOut` and `c(Byte) async[2] lossy from p.Out to q.In` respectively denote a synchronous and a lossy asynchronous channel named `c` that accepts messages with one `Byte` parameter, and the asynchronous channel may buffer up to two messages. A *port* is attached to at most one channel. Furthermore, messages sent over ports have a *name* and optionally a number of parameters with the same types as defined on the connected channel.

SLCO supports components at two levels: each object forms a component that can communicate with other objects via message-passing, while inside an object multiple components may exist that can interact via shared variables.

A state machine consists of local variables, a finite number of states, an initial state, and transitions between states. Transitions have an optional priority and a (possibly empty) sequence of *statements* associated with it. For instance, given user defined action `a` and variable `x`; the construct `1: s1 -> s2 {x := x + 1; a}` denotes a transition, with the priority 1, starting at state `s1` that first performs the statement `x := x + 1` and then performs the action `a`. Upon completion of the statements the state `s2` is reached. A lower number indicates higher priority. Higher priority transitions are considered for firing before lower priority ones. Transitions with the same priority are fired non-deterministically. By default, a transition has priority 0. Furthermore, a transition may be guarded by a boolean expression occurring as the transition's first statement; a guarded transition is able to fire iff the guard expression evaluates to **true**.

Parallel execution of transitions is formalised using an interleaving semantics, in which SLCO statements are atomic, i.e., the transition with sequence of statements is equivalent to a sequence of transitions each executing one of the statements in the same order. No finer-grained interleaving is allowed. SLCO offers five types of statements:

1. *(Boolean) Expression*: a condition that is blocked if it evaluates to **false**. In an expression, state machine-local and object-local variables may be referenced.

2. *Assignment*: `x := e` indicates that the evaluation of an expression `e` is assigned to variable `x`. The expression may be logical (boolean) or arithmetic expressions. Again, both state machine-local and object-local variables may be referenced. It is always able to fire.

3. *Composite*: a statement grouping an optional boolean expression and one or more assignments (in that order). It is enabled iff the expression at the head is enabled. If no expression is included, it is always able to fire. For instance, `[x>0; x:=x-1; y:=y+1]` (the square brackets denote a composite statement) indicates that in case `x` is greater than 0, `x` is decremented and `y` is incremented, all in one atomic step.

4. *Send* and *Receive*: `send` and `receive` attempt to send or receive a message to or from a particular channel, respectively. If the buffer associated to the channel

```
1  model Test {
2    actions init
3    classes
4      P { ... }
5      Q {
6        variables Integer x y
7        ports Out1 Out2 InOut
8        state machines
9        SM1 {
10         variables Boolean started:=false
11         initial Com0 states Com1 Com2
12         transitions
13           Com0 -> Com1 { send M(false,0) to Out1;
14                          started:=true }
15           Com1 -> Com1 { [x > 0; x:=x-1; y:=y+1];
16                          send N(y) to Out2}
17         1: Com1 -> Com2 { receive S() from InOut }
18           Com2 -> Com0 { init }
19       }
20       SM2 { ... }
21     }
22   objects p: P(), q: Q(x:=10, y:=0)
23   channels
24     c1(Boolean, Integer) async[2] lossless
25         from q.Out1 to p.In1
26     c2(Integer) async lossy from q.Out2 to p.In2
27     c3() sync between p.InOut and q.InOut
28 }
```

Figure 7.1: An example SLCO model

is full, a send operation is blocked. A receive operation is blocked if the buffer is empty, or if the next message is not as expected; the receive statement can store the received parameter values in variables, and check whether an expression related to these values evaluates to **true**. If not, the receive statement is not enabled, and communication fails.

5. *User-defined action*: an action that indicates yet-unspecified behaviour. User-defined actions can be implemented in code or transformed to concrete behaviour.

Figure 7.1 presents part of an SLCO model `Test`. It defines classes `P`, `Q` (lines 4, 5). At line 6-7 variables `x` and `y` and ports `Out1`, `Out2` and `InOut` are defined. Class `Q` contains state machines `SM1`, `SM2` (lines 9-20). At line 22, objects `p` and `q` are declared as instances of `P` and `Q`, respectively. The object ports are attached to channels at lines 24-27. Channel `c1` accepts messages with a `Boolean` parameter, is asynchronous, has a buffer of size 2, is lossless, and connects ports `Out1` and `In1` of `q` and `p`, respectively.

State machine `SM1` has a boolean local variable `started` (line 10), an initial state `Com0`, and other states `Com1` and `Com2` (line 11).

Between the states, transitions with sequences of atomic statements are defined (lines 12-18). The '1:' in front of the transition at line 17 denotes the *priority* of the transition. This priority enforces `SM1` in state `Com1` to consider the `Com1` self-loop (line 15) before the transition to `Com2` at line 17, i.e., it is first checked whether `x > 0`, only if this is not the case the transition to `Com2` is considered. Recall that messages optionally carry some parameters; a message named `M` with parameters **`false`** and `0` is sent at line 13, while the transition at line 17 attempts to receive a message named `S` without parameters. When the former transitions is fired the message `M` is stored in the buffer along with its parameters. The transition at line 17 is unable to fire as long as there are no state machines that are able to send message `S()`. At line 18, a user-defined action `init` is

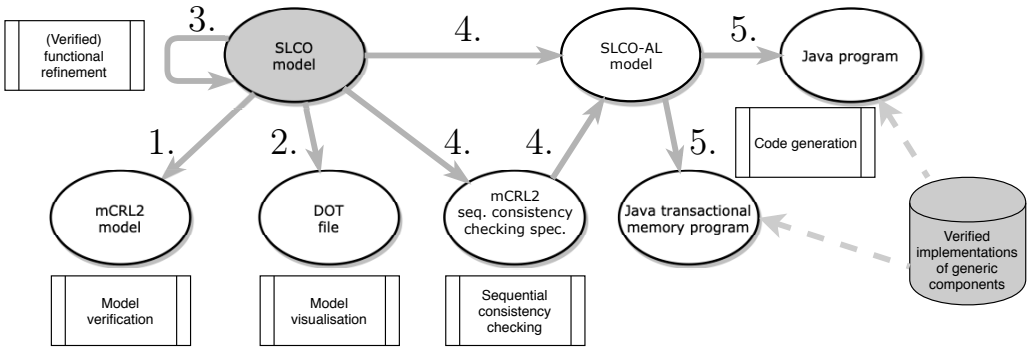


Figure 7.2: An overview of the SLCO framework

used, to indicate that some unspecified initialisation procedure is to be performed when moving from `Com2` to `Com0`.

7.4 Features of the Framework

Figure 7.2 provides an overview of the SLCO framework. The SLCO framework[2] is implemented in Python, using TEXTX [65] for meta-modelling and JINJA2[3] or HENSHIN [12] for model transformations. Given an SLCO model (the grey ellipse at the top-left corner), a number of features can be used:

1. Verification of the SLCO model via a translation to MCRL2. Compositional reasoning modulo Divergence-Preserving Branching Bisimulation (DPBB) is supported as a result of Chapter 4.

2. Model visualisation: graphical representation of the SLCO model via DOT.

3. SLCO model-to-model transformations using JINJA2 or HENSHIN. Some HENSHIN transformations can be verified (Chapter 3) via a translation of the HENSHIN transformation to REFINER rule systems [218].

4. Transformation to multi-threaded Java code using two verified code generators.

5. Reducing the use of synchronisation constructs through a translation to SLCO-AL (SLCO Annotated Level) relying on atomicity checking (Chapter 6) to detect how and where to synchronise.

1. Formal verification of SLCO models To formally verify that an SLCO model satisfies desirable functional properties, it can be transformed to an MCRL2 model. With the MCRL2 toolset [54], it is then possible to apply model checking [16]. Properties specified as μ-calculus formulas can be checked by first combining model and property into a Parametrised Boolean Equation System [93], and then checking the latter's state space.

Chapter 4 established that DPBB is a congruence for parallel composition and synchronisation. Hence, compositional abstraction modulo DPBB and branching bisimulation can be applied to MCRL2 models. The abstraction is applied globally to the MCRL2

[2]Git repository of the framework: `https://gitlab.tue.nl/SLCO/SLCO.git`.
[3]`http://jinja.pocoo.org`.

model hiding all actions that are not communicating actions or relevant for the property to be checked. This abstraction technique reduces the search space considered by mCRL2 and, therefore, allows faster verification and verification of larger models.

2. Model visualisation Slco models can be transformed to Dot files to visualise the state machines, thereby providing more insight into the structure of a model.

3. Slco model-to-model transformations Transformations can be used to iteratively re-factor or refine Slco models, for instance rewrite state machines or replace user-defined actions with concrete behaviour. Some user-defined transformations, specifically the ones between patterns of user-defined actions, can be verified directly for the preservation of functional properties, using our transformation verification technique [60] (Chapter 3) implemented in the Refiner tool [218]. It checks whether a transformation introduces patterns that are branching bisimilar to the replaced patterns after abstraction with respect to a given property. In other cases, preservation of properties can be determined for specific transformation applications by verifying the resulting Slco model via mCRL2.

Verification of transformations is done on transformations expressed in Henshin, a transformation language for the Eclipse Modelling Framework [193]. Henshin transformations can be translated into rule systems that are verified using Refiner. Currently, only transformations expressed over transitions with user-defined actions of a single state machine are supported. Supported transformations can be translated automatically to Refiner rule systems using our Henshin2Refiner tool. [4] These rule systems can then be verified by Refiner.

In Henshin transformations are expressed over patterns of object diagrams (i.e., an instance of a class diagram) of the source and target languages (in our case both are the Slco language). Formally a transformation consists of a match-pattern and a replace-pattern. The *match-pattern* must be present in any model of the source language that the transformation is applied to. The *replace-pattern* replaces every match of the match-pattern; i.e., the application of a transformation replaces every pattern that is matched on by the match-pattern by the *replace-pattern*. However, for convenience, Henshin displays both patterns in a single graph: links and elements that are present in both patterns, in the match pattern only, and in the replace pattern only are marked «preserve», «create», and «delete», respectively. As Refiner relies on the Double Push-out Approach, transformation rules in Henshin "Check Dangling" and "Injective Matching" must be set to **true** to ensure compatibility of the verification with the Henshin rule.

Figure 7.3 presents a transformation that refines user-defined actions named `init`; it inserts a transition performing a `processConfig` action after transitions with an `init` action. There are two transformation rules: *refineInit* (left) and addActionProcessConfig (right). The former rule performs the `processConfig` transition insertion, the latter rule has to be applied once to ensure well-formedness of the Slco model (i.e., it adds the `processConfig` action to the model if it is not already present).

The *addActionProcessConfig* rule matches any model that defines an `init` action (indicated by the two top most «preserve» nodes and link), but does not define a `processConfig` action (indicated by the «forbid» node and link at the bottom). If

[4] Available in the "SLCOtoSLCO_Verification/Henshin_to_REFINER" directory of the Slco Git repository

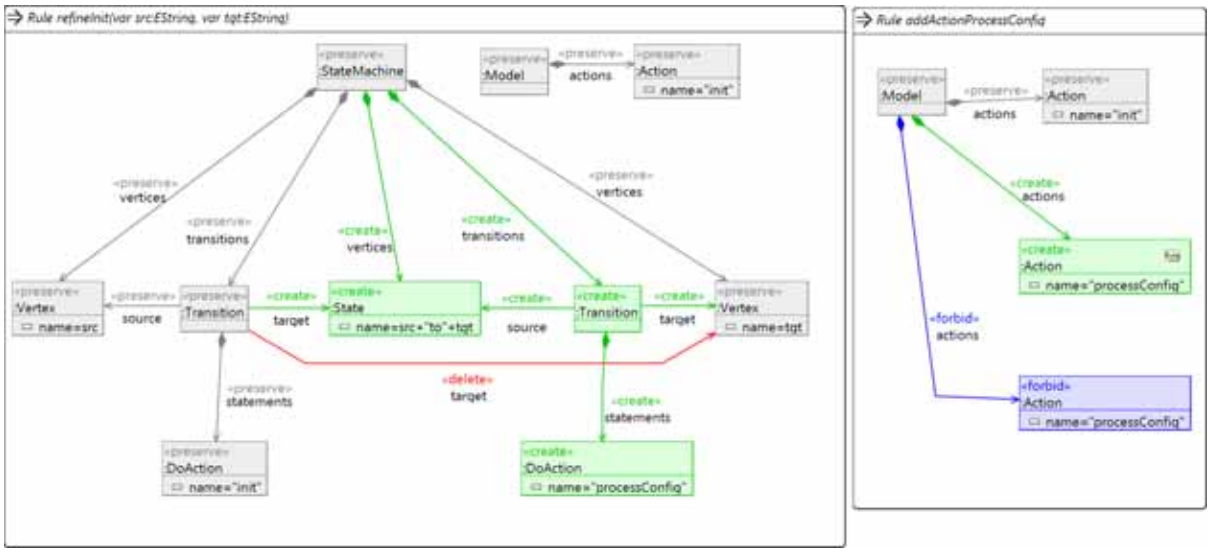


Figure 7.3: A Henshin transformation that refines user-defined actions named `init`; the left rule inserts a `processConfig` transition after an `init` transition, the right rule adds `processConfig` as a user-defined action if not already present

the rule matches, then the `processConfig` action is added to the model's user actions (indicated by the «create» node and link in the middle).

The `refineInit` rule matches any model that defines an `init` action (indicated by the two nodes at the top right) and one or more state machines containing one or more transitions with an `init` action (indicated by all remaining nodes that are preserved). The rule has two *var* parameters *src* and *tgt*; these variables are populated during the application of the transformation and are used to generate a new state named *src*+"`to`"+*tgt* (the middle node). The transformation replaces the source state of the original transition with the newly created stated (indicated by the «delete»-link between the right most node and the centre second left most node). Furthermore, a transition performing a `processConfig` action is created (indicated by «create» on the centre and bottom second right most nodes). All the nodes are appropriately linked to the state machine. If the *refineInit* rule is applied to the model defined in Figure 7.1, then transition `Com2 -> Com0 { init }` (line 18) will be replaced by two transitions `Com2 -> Com2toCom0 { processConfig }` and `Com2toCom0 -> Com0 { init }`.

Given a Henshin transformation over Slco the Henshin2Refiner tool translates the transformation to a Refiner rule system in three steps:

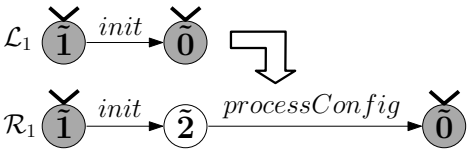
1. The match- and replace-patterns are extracted from the Henshin transformation and a mapping between the two patterns is created mapping the preserved nodes.

2. The two patterns are translated to patterns of Labelled Transition Systems (LTSs) following the Slco semantics. This step relies heavily on the way Slco semantics are specified. That is, semantics of incomplete models or patterns of models can be derived; e.g., the semantics of (part of) a state machine is formalised on an individual (state machine) basis allowing the specification of state machine patterns.

3. The generated LTS-patterns are written as a Refiner transformation rule and the rule system is completed by specifying that each action label in the LTSs operates independently of other concurrent processes or state machines.

```
1  transform
2    des(1,2,2,0)
3    (1,"init",0)
4    ->
5    des(2,3,2,0)
6    (1,"init",2)
7    (2,"processConfig",0)
8
9  add
10   "processConfig"
```

(a) The generated REFINER rule system



$\mathcal{V}' = \{ (\langle init \rangle, init) \}$

$\hat{\mathcal{V}} = \{ (\langle processConfig \rangle, processConfig) \}$

(b) The graphical representation of the generated REFINER rule system

Figure 7.4: The REFINER rule system generated by HENSHIN2REFINER from the HENSHIN transformation shown in Figure 7.3 and its graphical representation following the graphical representation presented in Chapter 3: the match-pattern is given by $\mathcal{L}_1$, the replace-pattern is given by $\mathcal{R}_1$, preserved states are indicated by an *incoming arrow* and *grey* colour, *white* states in $\mathcal{R}_1$ indicate states that are created, finally, $\mathcal{V}'$ and $\hat{\mathcal{V}}$ respectively contain 'to be matched' and 'to be added' synchronisation laws (in this figure the laws indicate that the `init` and `proccessConfig` are independent actions)

This methodology is not necessarily limited to transformations specified over SLCO: the methodology is applicable to any language that formalises the semantics in a compositional manner, such that LTSs over partial models can be derived. Hence, in the future, the HENSHIN2REFINER tool can be extended to other languages.

Figure 7.4 shows the REFINER rule system that is the result of the application of HENSHIN2REFINER to the HENSHIN transformation shown in Figure 7.3. The rule system's textual representation is given in Figure 7.4a and its graphical representation is given in Figure 7.4b. The *refineInit* rule is translated to the REFINER rule on lines 2-7 ($\mathcal{L}_1 \Rightarrow \mathcal{R}_1$). The match-pattern-LTS and replace-pattern-LTS are defined respectively at lines 2-3 ($\mathcal{L}_1$), and 5-7 ($\mathcal{R}_1$). The arrow -> tells REFINER that the LTS pattern above it is the match-pattern and the LTS below it is the replace-pattern. Furthermore, for the purpose of verification the `processConfig` action label is considered to be added in transformed LTSs (line 10 and $\hat{\mathcal{V}}$). The *addActionProcessConfig* rule is not translated since it does not concern dynamic semantics, but well-formedness. If the `processConfig` action is hidden before verification, then REFINER will indicate that this rule system is indeed dynamic semantics preserving.

4. Reducing the use of synchronisation constructs with SLCO-AL In implementation code, the naive use of nested locking or `atomic` blocks for *all* statements often leads to congestion and, thus, results in under-performing parallel programs. As previously mentioned, SLCO-AL can be used to instruct code generators. Currently, this only extends SLCO with a *fence* statement.

Some statements do not actually need protection by a synchronisation mechanism; in such a case, the lack of such synchronisations is not *observable*. Furthermore, it is possible that statements within a composite statement can be factored out in a way that the model remains *observably* SC with respect to the source model. To detect such situations, the framework provides a transformation to MCRL2 including the sequential consistency violation resolution algorithm based on the work on sequential consistency monitoring of parallel programs [62] in Chapter 6. This algorithm checks which specified data accesses in a model need to be protected in the code by a synchronisation mechanism in order to avoid potential *sequential consistency violations*. Furthermore, the algorithm determines

when a fence suffices as an alternative to the more heavy-weight locks or atomic blocks.

With the output of the sequential consistency monitor, composite statements can be decomposed and the need for synchronisations can be indicated in an SLCO-AL model. In the adapted model the use of these synchronisations is restricted to the absolutely necessary and least costly ones. The adapted model is semantically indistinguishable from the original one during execution of their respective generated code.

The transformation of SLCO to SLCO-AL proceeds as follows:

1. All composite statements are normalised. For each shared variable that is read from by at least two sub-statements a new statement is introduced that stores the shared variable locally, and the sub-statements are updated to refer to this local variable instead. Finally, all writes to a shared variable are localised similarly except for the last one. This normalisation is semantics preserving and ensures that there is at most one read and one write, respectively, from and to a shared variable: this satisfies with assumption of the algorithm.

2. The atomicity boundary on the composite statements is temporarily removed: the composite statement is converted to a regular statement sequence.

3. The statements are reordered according to the suggestion of the algorithm.

4. Fence suggestions by the algorithm are applied.

5. Composite statements are (re-)formed according to the suggestions of the algorithm.

This procedure strictly follows the suggestions of the sequential consistency monitor. Hence, the observable behaviour of the SLCO model the procedure is applied to remains unchanged.

5. Transformation to multi-threaded Java Before code is generated an SLCO model is translated to an SLCO-AL model. An SLCO-AL model is an SLCO model that is more specific on how and where to ensure atomicity.

The SLCO framework offers two partly verified code generators that take an SLCO-AL model and generate multi-threaded Java code. The generators use different methods to ensure atomicity of statements: the first generator uses a locking mechanism, while the second generator uses transactional memory.

In both generators, each state machine in the given SLCO-AL model is mapped to an individual thread. Hence, any variables shared by state machines correspond with shared variables in the Java code. The code is constructed modularly: implementations of generic concepts that are reusable in the generated code, such as channel and a locking mechanism for shared variables, have been added to a *generic component library* [224]. The functional correctness of these parts of the generator have been proven [33] using VERIFAST [107]: 1) the atomicity of statements is preserved in generated code, 2) messages sent over lossless channels are eventually received, and 3) generated code does not introduce deadlocks. In addition, the SLCO framework offers a verified robustness mechanism called *Failbox* [224] that is applied in the code to ensure that in case of a malfunctioning thread, dependent threads are notified if a thread fails.

The *first generator* enforces the use of a nested locking mechanism [225] to ensure that variables are safely shared. Each variable (and each array cell) is associated with an individual lock, and whenever for the execution of a statement a number of shared variables needs to be accessed, it is attempted to acquire the corresponding locks in a

```
1     ...
2   case Com1:
3     // [x > 0; x := x - 1; y := y + 1]
4     java_lockIDs[0] = 0; java_lockIDs[1] = 1;
5     java_kp.lock(java_lockIDs,2);
6     if (!(x > 0)) {
7       SignalMessage m = c3.receive("S");
8       java_kp.unlock(java_lockIDs,2);
9       if (!(m == null)) {
10        // Change state
11        java_currentState =
12          Test.java_State.Com2;
13      }
14      break;
15    }
16    x = x - 1; y = y + 1;
17    java_kp.unlock(java_lockIDs,2);
18    // send N(y) to Out2
19    java_lockIDs[0] = 1;
20    java_kp.lock(java_lockIDs,1);
21    c2.blocked_send("N", y);
22    java_kp.unlock(java_lockIDs,1);
23    // Change state
24    java_currentState = Test.java_State.Com1;
25    break;
26  case Com2:
27    ...
```

Figure 7.5: Excerpt of the generated Java implementation of the SLCO model in Figure 7.1

predefined order. The use of the fixed order prevents deadlocks and we have proven that it ensures the preservation of the atomicity of SLCO statements [225].

In Figure 7.5, part of the Java implementation of model `Test` of Figure 7.1 is presented. This part covers the transitions at lines 15-17 in the SLCO model and is part of a `switch` construct inside a `while` loop. This loop is responsible for the continuous movement between state machine states.

In the SLCO composite construct, shown at line 3, 6, and 16, class variables `x` and `y` are accessed. For this reason, locks need to be acquired for both variables before the statement can be executed. At line 4, the IDs for both variables are added to array `java_lockIDs` in a sorted way to ensure ordered locking. Finally, the locks are requested (line 5). If the locks are granted and the guard expression evaluates to `true`, the assignments of the composite statement are executed (line 16). Note the releasing of the locks once a statement has been executed. Alternatively, if the locks were not acquired or the guard expression evaluated to `false`, it is attempted to perform the `receive` statement specified at line 17 of the SLCO model. If the `receive` statement succeeded (line 9), the code 'changes state' according to the SLCO transition description (line 11). Finally, at line 18-23, the `send` statement at line 16 of the SLCO model is executed. It is executed as a possibly blocking send operation, since in the model, the state machine is in the middle of executing the statements of the transition at lines 15-16, and cannot consider alternatives.

The *second generator* enforces the use of transactional memory, relying on the ATOMJAVA code translation [100]. Instead of our nested locking mechanism, `atomic` blocks are used, to indicate that whenever the execution of a statement accesses a variable simultaneously accessed by another thread, the execution should be rolled back.

7.5 Roadmap

In the near future, we will continue our research in a number of directions. For instance, it is our goal that most, if not all, SLCO model transformations will be directly verifiable. Our current technique in REFINER [60, 218] is restricted to transformations between action patterns, as opposed to the transformation of (patterns of) other types of SLCO statements. Establishing that a transformation preserves properties for arbitrary input is stronger than having to verify resulting models each time the transformation is applied.

Regarding model verification, we plan to work on a new version of our GPU accelerated model checker GPUEXPLORE [221] that will accept SLCO models as input. Great speed-ups over 500× have been reported with this tool, and connecting SLCO will make it feasible to rapidly produce verification results for larger models. We will also continue our research on compositional model checking [58], to modularly verify SLCO models.

Regarding the SLCO-AL language, we will consider extending it to cover various other optimisation possibilities. In that respect, one can also think of optimising code with respect to other criteria than performance, such as power efficiency and security. To make smart decisions regarding quantitative characteristics of models, it may be required to extend our analysis towards probabilistic or stochastic model checking [16].

Research on verified code generation will focus on verifying complete programs, as opposed to only verifying generic components. We plan to use VERCORS [25] for this.

We plan to address the development of GPU software. For this, we need to extend SLCO to model such systems, and construct additional code generators.

Finally, we are considering to integrate our tool chain into the ECLIPSE IDE, to create one environment in which all tools in the framework can be accessed.

Chapter 8

Conclusions

In this chapter, first, we discuss the main contributions of the thesis. For each of the research questions presented in Chapter 1 we recall the main results and conclusions. Finally, directions for future work are discussed. Additional details are available in the chapters that cover the research questions.

8.1 Contributions

In this thesis we have investigated several areas of verification of concurrent systems: verification of model transformations, compositional minimisation of state spaces, selection of minimisation approaches, guaranteeing sequential consistency for generated code with respect to source models. Finally, we discussed how to integrate these in the Model-Driven Engineering (MDE) work flow. To this end, *five* research questions were formulated. Each of the research questions was addressed in one chapter of this thesis.

Verification of Transformations Current model transformation verification techniques are mainly focused on verifying preservation of well-formedness or static semantics [210]. So far little attention is given to the verification of dynamic semantics preservation for model transformations. Techniques that do support verification of dynamic semantics do not have intrinsic support for concurrency. The following research question addresses verification of transformations of concurrent systems.

> RQ 1:*How can we verify preservation of dynamic semantics of model transformations of concurrent models?*

This question is addressed in Chapter 3 where a verification method is developed for transformations of concurrent systems. A concurrent system is modelled as a network of Labelled Transition Systems (network of LTSs), or LTS network for short. A transformation is specified as a rule system consisting of a set of transformation rules over LTSs, a set of synchronisation laws that are expected in the input LTS network, and a

set of synchronisation laws that will be introduced. A transformation rule specifies an LTS pattern to match and its replacement.

A transformation is considered to be *correct* iff it preserves the dynamic semantics for all possible LTS networks it is applicable to. More specifically, the input and output networks of a transformation must be equivalent modulo branching bisimulation for any LTS network used as input. Formally, we have shown that given an LTS network $\mathcal{N}$ and a rule system Σ the transformed network $T_\Sigma(\mathcal{N})$ is branching bisimilar to $\mathcal{N}$ if the so called κ-extension of the composition of LTS patterns to match and the composition of the replacement LTS patterns are branching bisimilar and a small number of application conditions hold. The application conditions require that the transformation is aware of the synchronisation laws that are relevant for the correctness of the transformation.

Finally, the proof is mechanically verified using the Coq interactive theorem prover [1] and the verification method is tested in a number of experiments. The verification of transformations is extremely efficient as it only needs to consider the fraction of the state space of a system that is specified in the rule system. This makes the approach especially useful for the verification of details that would otherwise increase the state space to a size that is infeasible for conventional verification methods.

Compositional State Space Minimisation One of the greatest aids in the verification of concurrent systems is congruences for parallel composition operators that also perform synchronisations. For many equivalence relations it has been shown that they are a congruence for such a parallel composition operator. Although it is generally assumed that Divergence Preserving Branching Bisimulation (DPBB), the finest equivalence relation in the linear time - branching time spectrum of van Glabbeek [205], is also a congruence for parallel composition with synchronisation, no such results have been published. To this end we investigated the following research question.

RQ 2:*Is DPBB a congruence for parallel composition with synchronisations?*

Chapter 4 finally proves that DPBB is indeed a congruence for parallel composition with synchronisations. That is, we have shown that, given two LTS networks $\mathcal{N}$ and $\mathcal{N}'$ that are equivalent modulo DPBB and a third LTS network $\mathcal{P}$, $\mathcal{N}$ composed with $\mathcal{P}$ is equivalent modulo DPBB to M' composed with $\mathcal{P}$. Additionally, we show that DPBB is a congruence for LTS networks: given n LTSs $\mathcal{G}_1$ to $\mathcal{G}_n$ that are equivalent to LTSs $\mathcal{G}'_1$ to $\mathcal{G}'_n$, the LTS resulting from the composition of $\mathcal{G}_1$ to $\mathcal{G}_n$ is equivalent to the LTS resulting from the composition of $\mathcal{G}'_1$ to $\mathcal{G}'_n$ under the same set of synchronisation laws.

Sometimes, an existing network must be decomposed into several smaller ones. We have discussed how to decompose an existing specification of a concurrent system into sub-components that is consistent with the original specification. Furthermore, the components may be rearranged since parallel composition with synchronisation enjoys the desirable commutativity and associativity properties in the context of DPBB. Finally, confidence in the proofs was strengthened by mechanically verifying the most important proofs in Coq.

Selection of Minimisation Approaches The selection of verification techniques amongst the numerous alternatives is often a daunting task; it is often not obvious what technique offers the best (or even reasonable) performance for a given model. Frequently, the memory consumption of a verification technique is the major limiting factor for its

[1] `https://coq.inria.fr/`

application. The following research question considers this selection scenario for the compositional aggregation technique [55] as it is applicable in many scenarios concerning the model checking of concurrent systems.

> RQ 3: *When can compositional aggregation be expected to be more (memory) efficient than monolithic minimisation?*

This question is addressed in Chapter 5. We first analysed compositional aggregation on 119 subjects with varying topology, scale, and hiding set. The analysis was concluded with the following findings: 1) the amount of internal behaviour in process LTSs and the amount of synchronisation between process LTSs have the biggest impact on the performance; 2) the application of hiding may result in significant reduction of the state space and memory cost, the amount of hidden behaviour is less important as synchronisation will limit the effect of hiding; 3) among the five network topologies we considered, none of them fundamentally ruled out compositional aggregation as an effective technique; and 4) as the number of processes in an LTS network is increased, the effectiveness of compositional aggregation with respect to the monolithic approach tends to increase as well.

Furthermore, we applied a variation of machine learning methods to develop predictors for the effectiveness of two compositional aggregation heuristics. As data is scarce we resorted to data collected over to 1,615 generated models. The data was split in a training set, on which the machine learning methods were trained, and a test set, on which the learned predictors were validated.

The maximum memory cost and maximum number of generated transitions of compositional aggregation heuristics normalised with respect to monolithic minimisation was predicted at an average accuracy of one order of magnitude for half of the cases in our test set. The learned classification models that predict the best minimisation approach with respect to maximum memory cost and maximum number of generated transitions had an accuracy between 55% and 61% and between 68% and 74%, respectively, on the test set. The classifiers were most sensitive for the class of cases for which smart reduction was the best choice with a sensitivity value of 0.87 for maximum memory cost and 0.99 for maximum number of generated transitions. Metrics related to interleaving density and the number of transitions in an LTS were most important for regression techniques, while metrics related to hiding and interleaving density of sets of LTSs were found to be the most determining factor for classification techniques.

Guaranteeing Sequential Consistency of Concurrent Systems Most software developers expect their programs to be Sequentially Consistent (SC). A concurrent program is SC iff all operations of a program are executed in a total order, i.e., atomically and in the order specified by the program. Due to modern compiler and hardware optimisations non-SC behaviour may be observed during execution of a program. By appropriately using synchronisation mechanisms such as semaphores and atomic instructions one can ensure that a program is observably SC. The next research question was stated to investigate the preservation of (observable) sequential consistency for concurrent models.

> RQ 4: *How can sequential consistency be preserved (up to observation) from model to execution of generated code?*

In Chapter 6 we developed a monitor using a conflict graph that spots all non-SC execution traces of a concurrent program. Such non-SC traces are indicated by a cycle in

the conflict graph. Correctness of the monitor is underlined by the well known theory of Shasha and Snir [188]. The monitor was optimised to reduce its memory footprint by summarising the conflict graph. We have shown that this summarised conflict graph recognises the same cycles as the (normal) conflict graph, i.e., just like the conflict graph, the summarised conflict graph detects all non-SC execution traces.

The summarised conflict graph was further optimised for use in a model checker. The model checking algorithm monitors a set of memory accesses provided by a pre-processing step. Similar to the algorithm of Shasha and Snir, the model checker reports which accesses may cause SC violations and how they may possibly be resolved.

The algorithm is more precise than that of Sasha and Snir in certain aspects as it explores the dynamic semantics of the program, while it is less precise in other aspects as the model checker cannot limit its search to the critical cycles. Since the model checking algorithm can be focussed on a given set of accesses, the algorithm of Shasha and Snir can be used as preprocessing step to select a set of unsafe accesses.

Application of Results in MDE The results obtained from the previous research questions are applicable to some underlying mathematical model. The following research question addresses how to integrate these results in an MDE work flow to make them accessible for developers.

RQ 5:*How can our results be applied in an MDE context?*

Chapter 7 integrates the results of the previous research questions in the Simple Language for Communicating Objects (Slco) framework. Models in Slco describe concurrent state machines that may communicate with each other via message passing or shared variables.

In Slco model transformations are specified using Jinja2 or Henshin. Some Henshin transformations can be verified via a translation of the Henshin transformation to rule systems. These rule systems can then be verified using the approach that resulted from RQ 1.

The framework supports verification of Slco models via a translation to mCRL2. Compositional reasoning modulo Divergence-Preserving Branching Bisimulation (DPBB) is supported as a result of RQ 2.

SC execution of Slco models can be guaranteed as a result of RQ 4. An Slco model is transformed to an mCRL2 specification that includes the sequential consistency monitor. When the mCRL2 toolset explores its state space, it reports where synchronisation mechanisms need to be applied. Based on the findings of the model checker the Slco model is then translated to an Slco-al (Slco Annotated Level) model from which an efficient implementation can be generated that is indistinguishable from an SC one.

8.2 Future Work

In this section, we discuss some possible extensions of the work presented in this thesis.

Verification of Transformations There are a number of natural extensions for the transformation verification method proposed in this thesis.

To determine semantics preservation the verification method uses branching bisimulation equivalence. A natural extension would be to consider the DPBB equivalence relation

that supports distinction between deadlocks and livelocks. Furthermore, by combining the transformation verification method with the consistent decomposition presented in Chapter 4 the transformation verification technique can be made compositional as was proposed by Wijs [214]. The two results are a natural fit, thus, the correctness of compositional transformation verification can be proven by referencing these two results. Compositional transformation verification weakens the completeness condition ANC1 regarding synchronising behaviour being transformed (see Section 3.4.1). Therefore, the approach becomes applicable to systems with cyclic dependencies as well.

Timing information could be included in the LTSs to design timed systems and express transformations of timed behaviour. This would also introduce the possibility to analyse the impact a transformation will have on the performance of a system under transformation [219] by means of timed branching bisimulation checking [76]. The capability to reason about system performance could be further strengthened by also introducing probabilities on the LTS transitions [16]. Existing tools, such as PRISM [129] and extensions [32], could then be employed to conduct the analysis of the systems. An interesting challenge is then how to involve these probabilities in the verification of transformations as well.

Moreover, it would be interesting to investigate valorisation of the transformation verification approaches. To achieve this, the practical limitations of the pre-conditions of the method in industrial sized transformation systems must be examined first. Furthermore, the transformation verification method will need to be able to deal with variables with large or infinite data domains. Such a feature can be supported by integrating the verification approach with symbolic techniques for quotienting of Parametrised Boolean Equation Systems such as that of Neele, Willemse, and Groote [154].

Finally, we want to take into account the evolution of properties as the system under transformation becomes more detailed. We conjecture that this can be done by incorporating the property that holds before transformation in the context of the LTS patterns to match and replacement LTS patterns. The patterns should generically model a matched system in which the property holds. This further extends the flexibility of the verification approach as the current approach considers the most general context up to hiding property irrelevant behaviour.

Compositional State Space Minimisation The proof that DPPB is a congruence for parallel composition with synchronisations has been conducted in the context of LTS networks. An interesting direction for future work is the integration of the proof in a meta-theory for process algebra. This integration would give a straightforward extension of our results to parallel composition for process algebra formalisms.

Again, the consideration of time in compositional model checking would be an interesting topic for future work. Applying timed specification approaches such as that of Wijs and Fokkink [213, 219] with our results would allow to compositionally reason about timed behaviour.

Selection of Minimisation Approaches The work presented in this thesis has confirmed existing insights and revealed new insights through statistical means. The statistical power of the analysis performed should be further improved by including a larger amount of subjects in the study. Further study of aggregation orders could lead to the development of new heuristics for the selection of aggregation orders. As scalable models have now been thoroughly investigated, we can next focus on non-scalable models, of which many are publicly available.

For the classification and regression predictors we would like to strengthen the validation applying the predictors to a number of LTS networks provided by the literature. Moreover, the accuracy and precision of the predictors may be improved with more data. Therefore, we would like to apply the classification and regression algorithms to a larger data set.

The generation of realistic LTS networks also has many interesting directions for future work. We would like to investigate the use of commonly occurring graphlets in the generation of LTSs. Furthermore, we see potential in an alternative method for labelling the transitions of generated LTS where an unlabelled state space is constructed before labelling transitions of LTSs to ensure reachability of nearly all states in process LTSs.

We had to aggregate some of the metrics before we could apply machine learning techniques. The ability to apply machine learning techniques to graphs directly would open up many opportunities by preventing the need to aggregate graph data.

Guaranteeing Sequential Consistency of Concurrent Systems Scalability of the sequential consistency monitor in model checking is currently still a source of concern. However, we see potential to make the analysis more efficient. Techniques that iteratively search different parts of the state space may be applied (e.g., [220]). A similar approach could be very effective here, taking reported resolution for sequential consistency violations into account as the analysis continues. Efficiency can be further improved by applying partial order reduction methods specifically designed for transition systems labelled with memory accesses of instructions.

There are also other systems that need to appear SC such as GPU programs. Our approach could be particularly effective on GPUs as all threads execute the same thread program, and therefore try to perform the same set of memory accesses. It should suffice to consider only two threads when searching for possible sequential consistency violations.

The Slco framework The features of the Slco framework may be expanded in a number of directions. First, it is our goal that most, if not all, Slco model transformations will be directly verifiable. Our current transformation verification method is restricted to transformations between action patterns, as opposed to the transformation of (patterns of) other types of Slco statements. To support other types of statements the transformation verification method needs to be able to reason symbolically about the transformation of the state space.

So far we have not applied the predictors learned in Chapter 5. We would like to apply these predictors to select minimisation approaches for the compositional verification of Slco models.

Research on verified code generation will focus on verifying complete programs, as opposed to only verifying generic components. Moreover, as more and more software is written for GPU systems, we are considering further extension towards GPU programs.

Regarding the Slco-al language, we will consider extending it to cover various other optimisation possibilities. In that respect, one can also think of optimising code with respect to other criteria than performance, such as power efficiency and security. To make smart decisions regarding quantitative characteristics of models, it may be required to extend our analysis towards probabilistic or stochastic model checking [16].

Bibliography

[1] Abadi, M., and Lamport, L. The Existence of Refinement Mappings. *Theoretical Computer Science 82* (1991), 253–284.

[2] Abd Elkader, K., Grumberg, O., Păsăreanu, C. S., and Shoham, S. Automated Circular Assume-Guarantee Reasoning with N-way Decomposition and Alphabet Refinement. In *CAV* (2016), vol. 9779 of *LNCS*, Springer, pp. 329–351.

[3] Abrial, J.-R., and Abrial, J.-R. *The B-book: assigning programs to meanings*. Cambridge University Press, 2005.

[4] Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., and Voisin, L. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *STTT 12*, 6 (2010), 447–466.

[5] Adve, S. V. *Designing memory consistency models for shared-memory multiprocessors*, vol. 2. University of Wisconsin–Madison, 1993.

[6] Alglave, J., Maranget, L., Sarkar, S., and Sewell, P. Fences in Weak Memory Models. In *CAV* (2010), Springer Berlin Heidelberg, pp. 258–272.

[7] Amrani, M., Combemale, B., Lúcio, L., Selim, G. M. K., Dingel, J., Le Traon, Y., Vangheluwe, H., and Cordy, J. R. Formal Verification Techniques for Model Transformations: A Tridimensional Classification. *JOT 14*, 3 (2015), 1–43.

[8] Andersen, H. Partial Model Checking. In *LICS* (1995), IEEE Computer Society Press, pp. 398–407.

[9] Andersen, H. Partial Model Checking of Modal Equations: A Survey. *STTT 2*, 3 (1999), 242–259.

[10] Anderson, J. H., Kim, Y.-J., and Herman, T. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing 16*, 2-3 (2003), 75–110.

[11] Andova, S., van den Brand, M. G. J., and Engelen, L. Reusable and Correct Endogenous Model Transformations. In *ICMT* (2012), Springer Berlin Heidelberg, pp. 72–88.

[12] Arendt, T., Biermann, E., Jurack, S., Krause, C., and Taentzer, G. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *MODELS* (2010), Springer Berlin Heidelberg, pp. 121–135.

[13] Armstrong, B., and Eigenmann, R. Challenges in the automatic parallelization of large-scale computational applications. In *Commercial Applications for High-Performance Computing* (2001), vol. 4528, International Society for Optics and Photonics, pp. 50–61.

[14] ASCI. The Distributed ASCI Supercomputer DAS4. `http://www.cs.vu.nl/das4/`. Accessed: 2017-08-09.

[15] Avrunin, G. S., Corbett, J. C., Dwyer, M. B., Pasareanu, C. S., and Siegel, S. F. Comparing Finite-State Verification Techniques for Concurrent Software. Technical Report UM-CS-1999-069, University of Massachusetts, 1999.

[16] Baier, C., and Katoen, J.-P. *Principles of model checking*. MIT Press, 2008.

[17] Bal, H., Epema, D., de Laat, C., van Nieuwpoort, R., Romein, J., Seinstra, F., Snoek, C., and Wijshoff, H. A Medium-Scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *IEEE Computer 49*, 5 (May 2016), 54–63.

[18] Baldan, P., Corradini, A., Ehrig, H., Heckel, R., and König, B. Bisimilarity and Behaviour-preserving Reconfigurations of Open Petri Nets. In *CALCO* (2007), vol. 4624 of *LNCS*, Springer, pp. 126–142.

[19] Bernstein, P. A., and Goodman, N. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys 13*, 2 (June 1981), 185–221.

[20] Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development, Coq' Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[21] Bézivin, J., Breton, E., Dupé, G., and Valduriez, P. The ATL Transformation-based Model Management Framework. Research Report 03.08, IRIN, Université de Nantes, 2003.

[22] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic Model Checking without BDDs. In *TACAS* (1999), Springer Berlin Heidelberg, pp. 193–207.

[23] Blech, J. O., Glesner, S., and Leitner, J. Formal Verification of Java Code Generation from UML Models. In *3rd International Fujaba Days* (2005), Fujaba Days, pp. 49–56.

[24] Bliudze, S., and Sifakis, J. The Algebra of Connectors—Structuring Interaction in BIP. *IEEE Transactions on Computers 57*, 10 (Oct 2008), 1315–1330.

[25] Blom, S., Darabi, S., Huisman, M., and Oortwijn, W. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *IFM* (2017), vol. 10510 of *LNCS*, Springer, pp. 102–110.

[26] Bloom, B. Structural Operational Semantics for weak bisimulations. *Theoretical Computer Science 146*, 1 (1995), 25 – 68.

[27] Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M., and Wood, D. A. Performance Pathologies in Hardware Transactional Memory. In *ISCA* (2007), vol. 35 of *ACM SIGARCH Computer Architecture News*, ACM, pp. 81–91.

[28] Boehm, B., and Basili, V. R. Software Defect Reduction Top 10 List. *Computer 34*, 1 (Jan. 2001), 135–137.

[29] Boehm, H.-J., and Adve, S. V. Foundations of the C++ Concurrency Memory Model. In *PLDI* (2008), ACM, pp. 68–78.

[30] Bonferroni, C. Teoria statistica delle classi e calcolo delle probabilità. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze 8* (1936), 3–62.

[31] Bourke, T., Brun, L., Dagand, P.-E., Leroy, X., Pouzet, M., and Rieg, L. A Formally Verified Compiler for Lustre. In *PLDI* (2017), ACM SIGPLAN Notices, ACM, pp. 586–601.

[32] Bošnački, D., Edelkamp, S., Sulewski, D., and Wijs, A. GPU-PRISM: An Extension of PRISM for General Purpose Graphics Processing Units. In *PDMC* (2010), IEEE Computer Society Press, pp. 17–19.

[33] Bošnački, D., van den Brand, M., Gabriels, J., Jacobs, B., Kuiper, R., Roede, S., Wijs, A., and Zhang, D. Towards Modular Verification of Threaded Concurrent Executable Code Generated From DSL Models. In *FACS* (2015), vol. 9539 of *LNCS*, Springer, pp. 141–160.

[34] Bowen, J. P., and Hinchey, M. G. Formal Methods. In *Computer Science Handbook*. ACM, 2004, ch. 106, pp. 106–1–106–25.

[35] Bradfield, J., and Walukiewicz, I. *The mu-calculus and Model Checking*. Springer International Publishing, 2018, pp. 871–919.

[36] Braunstein, C., and Encrenaz, E. CTL-Property Transformation Along an Incremental Design Process. In *AVOCS* (2004), vol. 128 of *ENTCS*, Elsevier, pp. 263–278.

[37] Broadfoot, G. H. ASD Case Notes: Costs and Benefits of Applying Formal Methods to Industrial Control Software. In *FM* (2005), Springer Berlin Heidelberg, pp. 548–551.

[38] Burnim, J., Sen, K., and Stergiou, C. Sound and complete monitoring of sequential consistency for relaxed memory models. In *TACAS* (2011), Springer, pp. 11–25.

[39] Casanova, A. M. *The Concurrency Control Problem for Database Systems*. Springer, 1981.

[40] Cheung, S. C., and Kramer, J. Context Constraints for Compositional Reachability Analysis. *TOSEM 5*, 4 (Oct. 1996), 334–377.

[41] Chomicki, J., and Toman, D. Time in database systems. *Handbook of Temporal Reasoning in Artificial Intelligence* (2005), 429–467.

[42] Christen, M., Schenk, O., and Burkhart, H. PATUS: A Code Generation and Auto-Tuning Framework For Parallel Stencil Computations. In *ISPA* (2011), IEEE, pp. 676–687.

[43] Clarke, E. M., Emerson, E. A., Jha, S., and Sistla, A. P. Symmetry reductions in model checking. In *CAV* (1998), Springer Berlin Heidelberg, pp. 147–158.

[44] Clarke, E. M., Enders, R., Filkorn, T., and Jha, S. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design 9*, 1 (Aug 1996), 77–104.

[45] Clarke, E. M., Grumberg, O., and Peled, D. *Model Checking*. The MIT Press, 1999.

[46] Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification: LASER 2011* (2012), vol. 7682 of *LNCS*, Springer Berlin Heidelberg, pp. 1–30.

[47] Clarke, E. M., Long, D. E., and McMillan, K. L. Compositional model checking. In *LICS* (Jun 1989), IEEE Computer Society Press, pp. 353–362.

[48] Cleveland, W. S., and Devlin, S. J. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American statistical association 83*, 403 (1988), 596–610.

[49] Cobleigh, J. M., Avrunin, G. S., and Clarke, L. A. Breaking Up is Hard to Do: An Evaluation of Automated Assume-guarantee Reasoning. *TOSEM 17*, 2 (May 2008), 7:1–7:52.

[50] Cohen, J. Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit. *Psychological bulletin 70*, 4 (1968), 213.

[51] Combemale, B., Crégut, X., Garoche, P.-L., and Thirioux, X. Essay on Semantics Definition in MDE - An Instrumented Approach for Model Verification. *Journal of Software 4*, 9 (2009), 943–958.

[52] Cousot, P., and Cousot, R. Abstract Interpretation Frameworks. *Journal of Logic and Computation 2*, 4 (1992), 511–547.

[53] Cousot, P., and Cousot, R. Refining Model Checking by Abstract Interpretation. *Automated Software Engineering 6*, 1 (Jan. 1999), 69–95.

[54] Cranen, S., Groote, J., Keiren, J., Stappers, F., de Vink, E., Wesselink, W., and Willemse, T. An Overview of the mCRL2 Toolset and Its Recent Advances. In *TACAS* (2013), vol. 7795 of *LNCS*, Springer, pp. 199–213.

[55] Crouzen, P., and Hermanns, H. Aggregation ordering for massively compositional models. In *ACSD* (2010), IEEE, pp. 171–180.

[56] Crouzen, P., and Lang, F. *Smart Reduction*. Springer Berlin Heidelberg, 2011, pp. 111–126.

[57] de Moura, L., Ruess, H., and Sorea, M. Bounded Model Checking and Induction: From Refutation to Verification. In *CAV* (2003), Springer Berlin Heidelberg, pp. 14–26.

[58] de Putter, S., Lang, F., and Wijs, A. Compositional Model Checking is Lively - Extended Version. *Science of Computer Programming* (2019). Special Issue on FACS. *Manuscript under review.*

[59] de Putter, S., and Wijs, A. Verifying a Verifier: On the Formal Correctness of an LTS Transformation Verification Technique. In *FASE 2016* (2016), vol. 9633 of *LNCS*, Springer, pp. 383–400.

[60] de Putter, S., and Wijs, A. A formal verification technique for behavioural model-to-model transformations. *Formal Aspects of Computing* (Oct 2017).

[61] de Putter, S., and Wijs, A. To Compose, or Not to Compose, That Is the Question: An Analysis of Compositional State Space Generation. In *FM* (2018), Springer International Publishing, pp. 485–504.

[62] de Putter, S., and Wijs, A. Model Driven Avoidance of Atomicity Violations under Relaxed-Memory Models. In *ESOP* (2019). *Submitted.*

[63] de Putter, S., Wijs, A., and Zhang, D. The SLCO Framework for Verified, Model-Driven Construction of Component Software. In *FACS* (2018), LNCS, Springer, pp. 288–296.

[64] de Putter, S., and Wijs, A. J. Compositional Model Checking Is Lively. In *FACS* (2017), vol. 10487 of *LNCS*, Springer, pp. 117–136.

[65] Dejanović, I., Vaderna, R., Milosavljević, G., and Vuković, v. TextX: A Python tool for Domain-Specific Languages implementation. *Knowledge-Based Systems 115* (2017), 1–4.

[66] Deligiannis, P., Donaldson, A. F., Ketema, J., Lal, A., and Thomson, P. Asynchronous Programming, Analysis and Testing with State Machines. In *Proceedings of PLDI 2015* (2015), vol. 50 of *ACM SIGPLAN Notices*, ACM Press, pp. 154–164.

[67] Dice, D., and Shavit, N. Understanding Tradeoffs in Software Transactional Memory. In *CGO* (2007), IEEE Computer Society, pp. 21–33.

[68] Dodds, M., and Plump, D. Graph Transformation in Constant Time. In *ICGT* (2006), vol. 4178 of *LNCS*, Springer, pp. 367–382.

[69] Dormoy, F.-X. Scade 6: a model based solution for safety critical software development. In *ERTS* (2008), pp. 1–9.

[70] Emerson, E. A., and Halpern, J. Y. "Sometimes" and "Not Never" Revisited: On Branching Versus Linear Time Temporal Logic. *JACM 33*, 1 (Jan. 1986), 151–178.

[71] Engelen, L. *From Napkin Sketches To Reliable Software.* PhD thesis, Eindhoven University of Technology, 2012.

[72] Eppstein, D., Galil, Z., and Italiano, G. Dynamic Graph Algorithms. In *CRC Handbook of Algorithms and Theory of Computation*. CRC Press, 1997, ch. 22.

[73] Farzan, A., and Madhusudan, P. Monitoring Atomicity in Concurrent Programs. In *CAV* (2008), vol. 5123 of *LNCS*, Springer, pp. 52–65.

[74] Fernandez, J. *ALDEBARAN : un système de vérification par réduction de processus communicants. (Aldebaran : a system of verification of communicating processes by using reduction).* PhD thesis, Joseph Fourier University, Grenoble, France, 1988.

[75] Finger, M., and McBrien, P. Concurrency control for perceivedly instantaneous transactions in valid-time databases. In *TIME* (May 1997), pp. 112–118.

[76] Fokkink, W., Pang, J., and Wijs, A. Is Timed Branching Bisimilarity an Equivalence Indeed? In *FORMATS* (2005), vol. 3829 of *LNCS*, Springer, pp. 258–272.

[77] Foster, H., Uchitel, S., Magee, J., and J., K. Model-based verification of Web service compositions. In *ASE* (Oct 2003), pp. 152–161.

[78] Fürst, A., Hoang, T. S., Basin, D., Desai, K., Sato, N., and Miyazaki, K. Code Generation for Event-B. In *IFM* (2014), Springer International Publishing, pp. 323–338.

[79] Garavel, H. Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular. Research report, INRIA, 2007.

[80] Garavel, H., and Lang, F. SVL: A Scripting Language for Compositional Verification. In *FORTE* (2002), Kluwer, pp. 377–392.

[81] Garavel, H., Lang, F., and Mateescu, R. Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version). Research Report RR-8708, INRIA Grenoble - Rhône-Alpes, Apr. 2015.

[82] Garavel, H., Lang, F., Mateescu, R., and Serwe, W. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *TACAS* (March 2011), vol. 6605 of *LNCS*, Springer, pp. 372–387.

[83] Garavel, H., and Mounier, L. Specification and Verification of various Distributed Leader Election Algorithm for Unidirectional Ring Networks. Research Report RR-2986, INRIA, 1996.

[84] Garavel, H., and Sighireanu, M. A Graphical Parallel Composition Operator for Process Algebras. In *FORTE/PSTV* (1999), vol. 156 of *IFIP Conference Proceedings*, Kluwer, pp. 185–202.

[85] Gheorghiu Bobaru, M., Păsăreanu, C. S., and Giannakopoulou, D. Automated Assume-Guarantee Reasoning by Abstraction Refinement. In *CAV* (2008), Springer Berlin Heidelberg, pp. 135–148.

[86] Giese, H., Glesner, S., Leitner, J., Schäfer, W., and Wagner, R. Towards Verified Model Transformations. In *MoDeVVa* (2006), pp. 78–93.

[87] Giese, H., and Lambers, L. Towards Automatic Verification of Behavior Preservation for Model Transformation via Invariant Checking. In *ICGT* (2012), vol. 7562 of *LNCS*, Springer, pp. 249–263.

[88] Giese, H., and Wagner, R. Incremental Model Synchronization with Triple Graph Grammars. In *MODELS* (2006), Springer Berlin Heidelberg, pp. 543–557.

[89] Godefroid, P. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, 1994.

[90] Graf, S., and Steffen, B. Compositional minimization of finite state systems. In *CAV* (1991), Springer Berlin Heidelberg, pp. 186–196.

[91] Groote, J., Jansen, D., Keiren, J., and Wijs, A. An $O(m \log n)$ Algorithm for Computing Stuttering Equivalence and Branching Bisimulation. *ACM TOCL 18*, 2 (2017), 13:1–13:34.

[92] Groote, J. F., Kouters, T. W. D. M., and Osaiweran, A. Specification Guidelines to Avoid the State Space Explosion Problem. In *FSEN* (2012), Springer Berlin Heidelberg, pp. 112–127.

[93] Groote, J. F., and Willemse, T. Parameterised Boolean Equation Systems. In *CONCUR* (2004), Springer Berlin Heidelberg, pp. 308–324.

[94] Group, O. M. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*, 2016.

[95] Guillen-Scholten, J., Arbab, F., de Boer, F., and Bonsangue, M. A Channel-based Coordination Model for Components. *ENTCS 68*, 3 (2003), 419 – 438. FOCLASA (Satellite Workshop of CONCUR 2002).

[96] Gupta, A., McMillan, K. L., and Fu, Z. Automated Assumption Generation for Compositional Verification. In *CAV* (2007), Springer Berlin Heidelberg, pp. 420–432.

[97] Heimbold, D., and Luckham, D. Debugging Ada Tasking Programs. *IEEE Software 2* (1985), 47–57.

[98] Hennessy, M. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons, Inc., 1990.

[99] Hijma, P., Jacobs, C., van Nieuwpoort, R., and Bal, H. Cashmere: Heterogeneous Many-Core Computing. In *IPDPS* (2015).

[100] Hindman, B., and Grossman, D. Atomicity via Source-to-source Translation. In *MSPC* (2006), ACM Press, pp. 82–91.

[101] Hintze, J., and Nelson, R. Violin Plots: A Box Plot-Density Trace Synergism. *The American Statistician 52*, 2 (1998), 181–184.

[102] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Communications of the ACM 12*, 10 (Oct. 1969), 576–580.

[103] Hoare, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (Aug. 1978), 666–677.

[104] Holzmann, G. J. Formal methods for early fault detection. In *FTRTFT* (1996), Springer Berlin Heidelberg, pp. 40–54.

[105] Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., and Wehrheim, H. Showing Full Semantics Preservation in Model Transformations - A Comparison of Techniques. In *IFM* (2010), vol. 6396 of *LNCS*, Springer, pp. 183–198.

[106] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, 1989.

[107] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NFM* (2011), vol. 6617 of *LNCS*, Springer, pp. 41–55.

[108] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An introduction to statistical learning*, vol. 112. Springer, 2013.

[109] Jhala, R., and Majumdar, R. Software Model Checking. *ACM Computing Surveys 41*, 4 (Oct. 2009), 21:1–21:54.

[110] Jones, C. B. Specification and Design of (Parallel) Programs. In *IFIP congress* (1983), vol. 83, pp. 321–332.

[111] Kahlouche, H., Viho, C., and Zendri, M. An Industrial Experiment in Automatic Generation of Executable Test Suites for a Cache Coherency Protocol. In *IFIP TC6 11th International Workshop on Testing Communicating Systems* (1998), IWTCS, Kluwer, B.V., pp. 211–226.

[112] Kahsai, T., and Roggenbach, M. Property Preserving Refinement for Csp-Casl. In *WADT* (2008), vol. 5486 of *LNCS*, Springer, pp. 206–220.

[113] Karsai, G., and Narayanan, A. On the Correctness of Model Transformations in the Development of Embedded Systems. In *Monterey Workshop* (2007), vol. 4888 of *LNCS*, Springer, pp. 1–18.

[114] Keiren, J., Reniers, M. A., and Willemse, T. A. C. Structural Analysis of Boolean Equation Systems. *ACM TOCL 13*, 1 (2012), 8:1–8:35.

[115] Keller, R. K., Cameron, M., Taylor, R. N., and Troup, D. B. User Interface Development and Software Environments: The Chiron-1 System. In *ICSE* (1991), IEEE, pp. 208–218.

[116] Keller, R. M. Formal Verification of Parallel Programs. *Communications of the ACM 19*, 7 (July 1976), 371–384.

[117] Kendall, M., and Gibbons, J. *Rank correlation methods*, 5 ed. Oxford University Press, 1990, ch. 3.

[118] Keshishzadeh, S., and Mooij, A. J. Formalizing and testing the consistency of DSL transformations. *Formal Aspects of Computing 28*, 2 (Apr 2016), 181–206.

[119] Kleppe, A., Warmer, J., and Bast, W. *MDA Explained: The Model Driven Architecture(TM): Practice and Promise*. Addison-Wesley Professional, 2005.

[120] Kohavi, R. A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection. In *IJCAI* (1995), Morgan Kaufmann Publishers Inc., pp. 1137–1143.

[121] Kolovos, D. S., Paige, R. F., and Polack, F. A. C. The Epsilon Transformation Language. In *ICMT* (2008), Springer Berlin Heidelberg, pp. 46–60.

[122] Koomen, C. J. Algebraic specification and verification of communication protocols. *Science of Computer Programming 5* (1985), 1 – 36.

[123] Kozen, D. Results on the Propositional μ-calculus. *Theoretical Computer Science 27* (1983), 333–354.

[124] Krimm, J.-P., and Mounier, L. *Compositional state space generation from LOTOS programs*. Springer, 1997, pp. 239–258.

[125] Kuhn, M. Caret package. *JSS 28*, 5 (2008), 1–26.

[126] Kuhn, M., and Johnson, K. *Applied predictive modeling*, vol. 26. Springer, 2013.

[127] Kundu, S., Lerner, S., and Gupta, R. Automated Refinement Checking of Concurrent Systems. In *ICCAD* (2007), IEEE, pp. 318–325.

[128] Kuperstein, M., Vechev, M., and Yahav, E. Automatic Inference of Memory Fences. *SIGACT News 43*, 2 (June 2012), 108–123.

[129] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV* (2011), vol. 6806 of *LNCS*, Springer, pp. 585–591.

[130] Laboratory for Advanced Software Engineering Research. Example Repository for Finite State Verification Tools. `http://laser.cs.umass.edu/verification-examples/`, 8 Jan. 2003. Last Accessed 20 Dec. 2017.

[131] Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers 28*, 9 (Sept. 1979), 690–691.

[132] Lang, F. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In *IFM* (2005), vol. 3771 of *LNCS*, Springer, pp. 70–88.

[133] Lang, F. Refined Interfaces for Compositional Verification. In *FORTE* (2006), vol. 4229 of *LNCS*, Springer, pp. 159–174.

[134] Lang, F. Unpublished textual and PVS proof stating that branching bisimulation is a congruence for Networks of LTSs, this proof does not consider divergence-preserving branching bisimulation, 2016. Personal communication.

[135] Lang, F., and Mateescu, R. Partial Model Checking Using Networks of Labelled Transition Systems and Boolean Equation Systems. In *TACAS* (2012), Springer Berlin Heidelberg, pp. 141–156.

[136] Lang, F., and Mateescu, R. Partial Model Checking Using Networks of Labelled Transition Systems and Boolean Equation Systems. *Logical Methods in Computer Science 9*, 4 (2013), 1–32.

[137] Lano, K. *The B Language and Method, A Guide to Practical Formal Development*. Springer, 1996.

[138] Le Lann, G. Distributed Systems - Towards a Formal Approach. In *IFIP Congress* (1977), pp. 155–160.

[139] Long, D. E. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon University, 1993. UMI Order No. GAX94-02579.

[140] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS* (2008), pp. 329–339.

[141] Luttik, S. P. Description and Formal Specification of the Link Layer of P1394. Technical Report SEN-R9706, CWI, 1997.

[142] Manson, J., Pugh, W., and Adve, S. V. The Java Memory Model. In *POPL* (2005), ACM, pp. 378–391.

[143] Maraninchi, F. *Operational and compositional semantics of synchronous automaton compositions*. Springer, 1992, pp. 550–564.

[144] Mateescu, R., and Wijs, A. Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. *Science of Computer Programming 96*, 3 (2014), 354–376.

[145] Mazzara, M., and Lanese, I. Towards a Unifying Theory for Web Services Composition. In *WS-FM* (2006), vol. 4184 of *LNCS*, Springer, pp. 257–272.

[146] McMillan, K. L. *Symbolic Model Checking*. Springer US, 1993, pp. 25–60.

[147] McMillan, K. L. Interpolation and SAT-Based Model Checking. In *CAV* (2003), Springer Berlin Heidelberg, pp. 1–13.

[148] Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.

[149] Milner, R., Parrow, J., and Walker, D. A Calculus of Mobile Processes, I & II. *Information and Computation 100*, 1 (Sept. 1992), 1–77.

[150] Mohagheghi, P., Gilani, W., Stefanescu, A., Fernandez, M. A., Nordmoen, B., and Fritzsche, M. Where does model-driven engineering help? experiences from three industrial cases. *Software & Systems Modeling 12*, 3 (Jul 2013), 619–639.

[151] Mounier, L. A LOTOS Specification of a "Transit-Node". Technical Report SPECTRE 94-8, VERIMAG, 3 1994.

[152] Narayanan, A., and Karsai, G. Towards Verifying Model Transformations. In *GT-VMT* (2008), vol. 211 of *ENTCS*, Elsevier, pp. 191–200.

[153] National Institute of Standards and Technology. *Data Encryption Standard (DES). Federal Information Processing Standards 46-3*, 1999.

[154] Neele, T., Willemse, T., and Groote, J. F. Solving Parameterised Boolean Equation Systems with Infinite Data Through Quotienting. In *FACS* (2018), LNCS, Springer International Publishing, pp. 216–236.

[155] Nicola, R., and Vaandrager, F. Action versus State Based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes, LITP Spring School on Theoretical Computer Science* (1990), vol. 469 of *LNCS*, Springer, pp. 407–419.

[156] Object Management Group. *Object Management Group Model Driven Architecture (MDA) MDA Guide*, 2014.

[157] O'Halloran, C. Automated verification of code automatically generated from Simulink®. *Automated Software Engineering 20*, 2 (Jun 2013), 237–264.

[158] O'Leary, Z. *The Essential Guide to Doing Research*. SAGE Publications, 2004.

[159] Papadimitriou, C. *The Theory of Database Concurrency Control*. Computer Science Press, Inc., 1986.

[160] Păsăreanu, C. S., Dwyer, M. B., and Huth, M. Assume-Guarantee Model Checking of Software: A Comparative Case Study. In *Theoretical and Practical Aspects of SPIN Model Checking* (1999), Springer Berlin Heidelberg, pp. 168–183.

[161] Păsăreanu, C. S., Giannakopoulou, D., Bobaru, M. G., Cobleigh, J. M., and Barringer, H. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design 32*, 3 (Jun 2008), 175–205.

[162] Pelánek, R. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN* (2007), vol. 4595 of *LNCS*, Springer, pp. 263–267.

[163] Pelánek, R. Properties of state spaces and their applications. *STTT 10*, 5 (Oct 2008), 443–454.

[164] Peled, D. All from one, one for all: on model checking using representatives. In *CAV* (1993), Springer, pp. 409–423.

[165] Peled, D. *Ten years of partial order reduction*. Springer, 1998, pp. 17–28.

[166] Peled, D. *Partial-Order Reduction*. Springer International Publishing, 2018, pp. 173–190.

[167] Peterson, G. L. Myths About the Mutual Exclusion Problem. *Information Processing Letters 12* (1981), 115–116.

[168] Petri, C. A. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.

[169] Ploeger, B. Analysis of ACS Using mCRL2. Technical Report 09-11, Eindhoven University of Technology, 2009.

[170] Plotkin, G. D. The origins of Structural Operational Semantics. *The Journal of Logic and Algebraic Programming 60-61* (2004), 3 – 15. Structural Operational Semantics.

[171] Pnueli, A. In Transition From Global to Modular Temporal Reasoning about Programs. In *Logics and Models of Concurrent Systems* (1985), vol. 13 of *NATO ASI*, Springer, pp. 123–144.

[172] Poetzl, D., and Kroening, D. Formalizing and Checking Thread Refinement for Data-Race-Free Execution Models. In *TACAS* (2016), vol. 9636 of *LNCS*, Springer, pp. 515–530.

[173] Rahim, L. A., and Whittle, J. A survey of approaches for verifying model transformations. *Software and Systems Modeling* (2013), 1–26.

[174] Ramalingam, G., and Reps, T. On the Computational Complexity of Dynamic Graph Problems. *Theoretical Computer Science 158* (1996), 233–277.

[175] Rivera, V., and Cataño, N. Code generation for Event-B. *STTT 19*, 1 (2017), 31–52.

[176] Rodriguez, J. D., Perez, A., and Lozano, J. A. Sensitivity Analysis of k-Fold Cross Validation in Prediction Error Estimation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 32*, 3 (March 2010), 569–575.

[177] Romijn, J. Model Checking a HAVi Leader Election Protocol. Technical Report SEN-R9915, CWI, 1999.

[178] Rompf, T., and Odersky, M. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. *Communications of the ACM 55*, 6 (2012), 121–130.

[179] Roscoe, A. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[180] Roscoe, A. W. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1997.

[181] Rünger, G. Parallel Programming Models for Irregular Algorithms. In *Parallel Algorithms and Cluster Computing* (2006), Springer Berlin Heidelberg, pp. 3–23.

[182] Rupert Jr, G. *Simultaneous statistical inference*. Springer Science & Business Media, 2012.

[183] Sabnani, K. K., Lapone, A. M., and Uyar, M. U. An algorithmic procedure for checking safety properties of protocols. *IEEE Transactions on Communications 37*, 9 (1989), 940–948.

[184] Saha, D. An Incremental Bisimulation Algorithm. In *FSTTCS* (2007), vol. 4855 of *LNCS*, Springer, pp. 204–215.

[185] Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *Computer 39*, 2 (Feb 2006), 25–31.

[186] Schürr, A. Specification of graph translators with Triple Graph Grammars. In *WG* (1994), Springer, pp. 151–163.

[187] Selim, G., Lúcio, L., Cordy, J., Dingel, J., and Oakes, B. Specification and Verification of Graph-Based Model Transformation Properties. In *ICGT* (2014), vol. 8571 of *LNCS*, Springer, pp. 113–129.

[188] Shasha, D., and Snir, M. Efficient and Correct Execution of Parallel Programs That Share Memory. *TOPLAS 10*, 2 (Apr. 1988), 282–312.

[189] Shavit, N., and Touitou, D. Software Transactional Memory. *Distributed Computing 10*, 2 (1997), 99–116.

[190] Silva, M. 50 years after the PhD thesis of Carl Adam Petri: A perspective. *IFAC Proceedings Volumes 45*, 29 (2012), 13 – 20. 11th IFAC Workshop on Discrete Event Systems.

[191] Sokolsky, O., and Smolka, S. Incremental Model Checking in the Modal Mu-Calculus. In *CAV* (1994), vol. 818 of *LNCS*, Springer, pp. 351–363.

[192] Spaninks, L. An Axiomatisation for Rooted Branching Bisimulation with Explicit Divergence. Master's thesis, Eindhoven University of Technology, 2013.

[193] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008.

[194] Stenzel, K., Moebius, N., and Reif, W. Formal Verification of QVT Transformations for Code Generation. In *MODELS* (2011), vol. 6981 of *LNCS*, Springer, pp. 533–547.

[195] Swamy, G. M. *Incremental Methods for Formal Verification and Logic Synthesis*. PhD thesis, University of California, 1996.

[196] Tai, K.-C., and Koppol, P. V. Hierarchy-based incremental analysis of communication protocols. In *ICNP* (1993), IEEE, pp. 318–325.

[197] Tai, K.-C., and Koppol, P. V. An incremental approach to reachability analysis of distributed programs. In *IWSSD* (1993), IEEE Computer Society Press, pp. 141–150.

[198] Talupur, M. Hardware Model Checking: Status, Challenges, and Opportunities. In *FMCAD* (2011), FMCAD Inc, pp. 154–154.

[199] Tarjan, R. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing 1*, 2 (1972), 146–160.

[200] The MathWorks Inc. Simulink®.

[201] Ulidowski, I., and Phillips, I. Ordered SOS Process Languages for Branching and Eager Bisimulations. *Information and Computation 178*, 1 (2002), 180 – 213.

[202] Valmari, A. Stubborn Sets for Reduced State Space Generation. In *Advances in Petri Nets* (1990), vol. 483 of *LNCS*, Springer, pp. 491–515.

[203] Valmari, A. Compositional state space generation. In *Advances in Petri Nets* (1993), Springer, pp. 427–457.

[204] van Glabbeek, R., Luttik, S., and Trčka, N. Branching Bisimilarity with Explicit Divergence. *Fundamenta Informaticae 93*, 4 (Dec. 2009), 371–392.

[205] van Glabbeek, R. J. The linear time — Branching time spectrum II. In *CONCUR* (1993), Springer Berlin Heidelberg, pp. 66–81.

[206] van Glabbeek, R. J., Luttik, B., and Trčka, N. Computation Tree Logic with Deadlock Detection. *LMCS 5*, 4 (2009).

[207] van Glabbeek, R. J., and Weijland, W. P. Branching Time and Abstraction in Bisimulation Semantics. *JACM 43*, 3 (1996), 555–600.

[208] Varró, D., and Pataricza, A. Automated Formal Verification of Model Transformations. In *CSDUML* (Sept. 2003), pp. 63–78.

[209] Verhoef, C. *A congruence theorem for structured operational semantics with predicates and negative premises.* Springer, 1994, pp. 433–448.

[210] Whittle, J., Hutchinson, J., and Rouncefield, M. The State of Practice in Model-Driven Engineering. *IEEE Software 31*, 3 (May 2014), 79–85.

[211] Wijs, A., and Bošnački, D. Many-core on-the-fly model checking of safety properties using GPUs. *STTT 18*, 2 (Apr 2016), 169–185.

[212] Wijs, A., Katoen, J.-P., and Bošnački, D. Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. *Formal Methods in System Design 48*, 3 (2016), 274–300.

[213] Wijs, A. J. Achieving Discrete Relative Timing with Untimed Process Algebra. In *ICECCS* (2007), IEEE Computer Society Press, pp. 35–44.

[214] Wijs, A. J. Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In *FACS* (2013), vol. 8348 of *LNCS*, Springer, pp. 348–368.

[215] Wijs, A. J. Confluence Detection for Transformations of Labelled Transition Systems. In *GaM* (2015), vol. 181 of *EPTCS*, Open Publishing Association, pp. 1–15.

[216] Wijs, A. J. GPU Accelerated Strong and Branching Bisimilarity Checking. In *TACAS* (2015), vol. 9035 of *LNCS*, Springer, pp. 368–383.

[217] Wijs, A. J., and Engelen, L. J. P. Efficient Property Preservation Checking of Model Refinements. In *TACAS* (2013), vol. 7795 of *LNCS*, Springer, pp. 565–579.

[218] Wijs, A. J., and Engelen, L. J. P. Refiner: Towards Formal Verification of Model Transformations. In *NFM 2014* (2014), vol. 8430 of *LNCS*, Springer, pp. 258–263.

[219] Wijs, A. J., and Fokkink, W. J. From χ_t to μCRL: Combining Performance and Functional Analysis. In *ICECCS* (2005), IEEE Computer Society Press, pp. 184–193.

[220] Wijs, A. J., and Neele, T. Compositional Model Checking with Incremental Counter-Example Construction. In *CAV* (2017), vol. 10426 of *LNCS*, Springer, pp. 570–590.

[221] Wijs, A. J., Neele, T., and Bošnački, D. GPUexplore 2.0: Unleashing GPU Explicit-state Model Checking. In *FM* (2016), vol. 9995 of *LNCS*, Springer, pp. 694–701.

[222] Winskel, G. A Compositional Proof System on a Category of Labelled Transition Systems. *Information and Computation 87*, 1-2 (1990), 2–57.

[223] Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H. Towards Automatic Model Synchronization from Model Transformations. In *ASE* (2007), ACM, pp. 164–173.

[224] Zhang, D. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. PhD thesis, Eindhoven University of Technology, 2018.

[225] Zhang, D., Bošnački, D., van den Brand, M., Huizing, C., Jacobs, B., Kuiper, R., and Wijs, A. Verifying Atomicity Preservation and Deadlock Freedom of a Generic Shared Variable Mechanism Used in Model-To-Code Transformations. In *MODELSWARD* (2017), vol. 692 of *CCIS*, Springer, pp. 249–273.

[226] Zhang, D., Bošnački, D., van den Brand, M., Huizing, C., Jacobs, B., Kuiper, R., and Wijs, A. Verification of Atomicity Preservation in Model-To-Code Transformations. In *MODELSWARD* (2016), SCITEPRESS, pp. 578–588.

[227] Zheng, L., Liao, X., Wu, S., Fan, X., and Jin, H. Understanding and Identifying Latent Data Races Cross-Thread Interleaving. *Frontiers of Computer Science 9*, 4 (2015), 524–539.

Summary

Verification of Concurrent Systems in a Model-Driven Engineering Workflow

Concurrent systems form an integral part of today's society. From smartphones, desktops and web systems to the car you drive, and even your coffee machine, concurrent systems can be found everywhere. Concurrency in systems has many benefits; e.g. better performance, better distribution of services. However, due to their non-deterministic nature concurrent systems are also more complex, harder to understand, and harder to develop than sequential programs.

Researchers and practitioners have sought to alleviate complexity, increase understandability, and facilitate the early and automated detection of faults. To this end, formal methods and Model-Driven Engineering (MDE) are widely applied.

In this thesis, we investigate automated formal methods to determine and guarantee correctness of concurrent systems and the integration of formal methods with MDE.

Over the years, various formal methods have been proposed and further developed to determine the functional correctness of concurrent systems. One of those methods is called model-checking in which a model is verified as a formal representation of the concurrent system against a number of requirements. The requirements are expressed as formal properties of the model. The model-checker then determines whether the formal behaviour described by de model, also called the state-space, satisfies these formal properties. Thus, model-checking gives guarantees that a model meets the specified requirements. With the model serving as a formal specification of the concurrent system, the expectation is that the implementations of the concurrent system will also meets the requirements. Yet there are several significant problems to cover.

First, if the implementation does not formally adhere to the specification all guarantees are void. The closer the model serving as specification is to the actual implementation, the more confidence one has in the correctness of the implementation.

A rigorous solution to the this problem is to generate code that is proven to preserve desired properties. However, the verified specification models from which this code must be generated are often lacking important details due to the high abstraction level. To bridge the gap between specification and implementation a series of transformations can be applied adding the required details. These transformations have to be verified as well. Nonetheless, verification of model transformations is still in its infancy.

In this thesis we propose a formal verification technique to determine that formalisations of such transformations in the form of rule systems are guaranteed to preserve functional properties, regardless of the models they are applied on.

Second, another problem arises when the model becomes too detailed called the state space explosion problem. The overall state space of a concurrent system is exponential in the size of the individual components and the number of parallel components. There are many model checking techniques making use of the fact that certain equivalence relations are congruences for parallel composition with synchronisation to reduce the state space explosion.

Since Divergence Preserving Branching Bisimulation (DPBB) is the finest equivalence relation in the linear time – branching time spectrum of van Glabbeek, it is desirable to apply DPBB as a first state space minimisation step. Although it is generally assumed that DPBB is a congruence for parallel composition with synchronisation, no proof has been published. This thesis finally proves that this is indeed the case.

Third, to further limit state space explosion, several approaches have been proposed that reason compositionally about concurrent systems. One such approach, called compositional aggregation, iteratively reduces the concurrent components modulo an appropriate equivalence relation during the construction of the state space. Compositional aggregation has shown to perform better (in the size of the largest state space in memory at one time) than classical monolithic composition in a number of cases. Although, there are also cases in which compositional aggregation performs much worse. It is unclear when one should apply compositional aggregation in favour of other techniques and how it is affected by action hiding and the scale of the model.

This thesis presents a descriptive analysis following the quantitative experimental approach and apply machine learning techniques to predict which of three compositional strategies performs best in terms of maximum memory cost. The learned prediction models achieved an accuracy between 55% and 61% on unseen data.

Fourth, even when code is generated from the verified model, bugs may still occur due to the interleaving of parallel execution of the components. Software developers expect their executions of their programs to be Sequentially Consistent (SC), i.e., the program executes in sequential order and atomic behaviour is not interrupted. Violations of sequential consistency are a major source of bugs. These can be avoided by appropriately using synchronisation mechanisms such as fences, semaphores and atomic instructions. On the other hand, over using such mechanisms can negatively impact performance.

Static analysis techniques can be used to insert synchronisation mechanisms that guarantee that program executions are indistinguishable SC ones. Alternatively, executions can be monitored and violations can be reported at run-time. While the first approach is more efficient (no run-time monitoring is required), the second approach is more precise.

In this thesis we combine the two approaches into an efficient monitor for model checkers. The output of the algorithm is used to add annotations to the concurrent model. These annotations are then used to generate efficient code that employs a minimal set of synchronisation mechanisms.

Finally, we combine our results in an framework supporting verification various aspects of the MDE workflow. The framework is centred around a domain specific language for concurrent state machine models. The framework support verification of models and model transformations, and offers a verified code generated. The generator produces efficient implementations of which executions are observably SC with respect to the corresponding model.

Curriculum Vitae

Personal Information

Name: Sander M. J. de Putter
Date of birth: March 10, 1986
Place of birth: Oostburg, the Netherlands

Education

Ph.D Candidate 2014.9–2019.1

Eindhoven University of Technology, Eindhoven, the Netherlands

M.Sc in Computer Science and Engineering 2011.9–2014.6

Eindhoven University of Technology, Eindhoven, the Netherlands

B.Sc in Technische Informatica 2003.9–2007.7

Hogeschool Avans, Breda, the Netherlands

Work Experience

Software Engineer 2007.9–2011.7

VSTEP B.V, Rotterdam, the Netherlands

Software Engineer Intern 2007.2–2007.7

VSTEP B.V, Rotterdam, the Netherlands

Software Engineer Intern 2005.9–2006.2

Codeglue, Schiedam, the Netherlands

Research Project Experience

AIPP5 EMC2 project (Embedded Multi-Core systems for Mixed Criticality applications in dynamic and changeable real-time environments) 2014.9–2017.6

Eindhoven University of Technology, Eindhoven, the Netherlands

Awards

Best Student Paper Award 2017.10

Formal Aspects of Component Software (FACS) 2017, Braga, Portugal

Extracurricular Activities

Organiser of the colloquia of the Software Engineering Technology group 2014.10–2018.1

Eindhoven University of Technology, Eindhoven, the Netherlands

Member of the PhD council of Mathematics and Computer Science 2015.5–2016.9

Eindhoven University of Technology, Eindhoven, the Netherlands

Member of the student sounding board of the department of informatica 2004.9–2005.7

Hogeschool Avans, Breda, the Netherlands

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World.* Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction.* Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems.* Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness.* Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle.* Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking.* Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography.* Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems.* Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars.* Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology.* Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction.* Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things.* Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes.* Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq.* Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

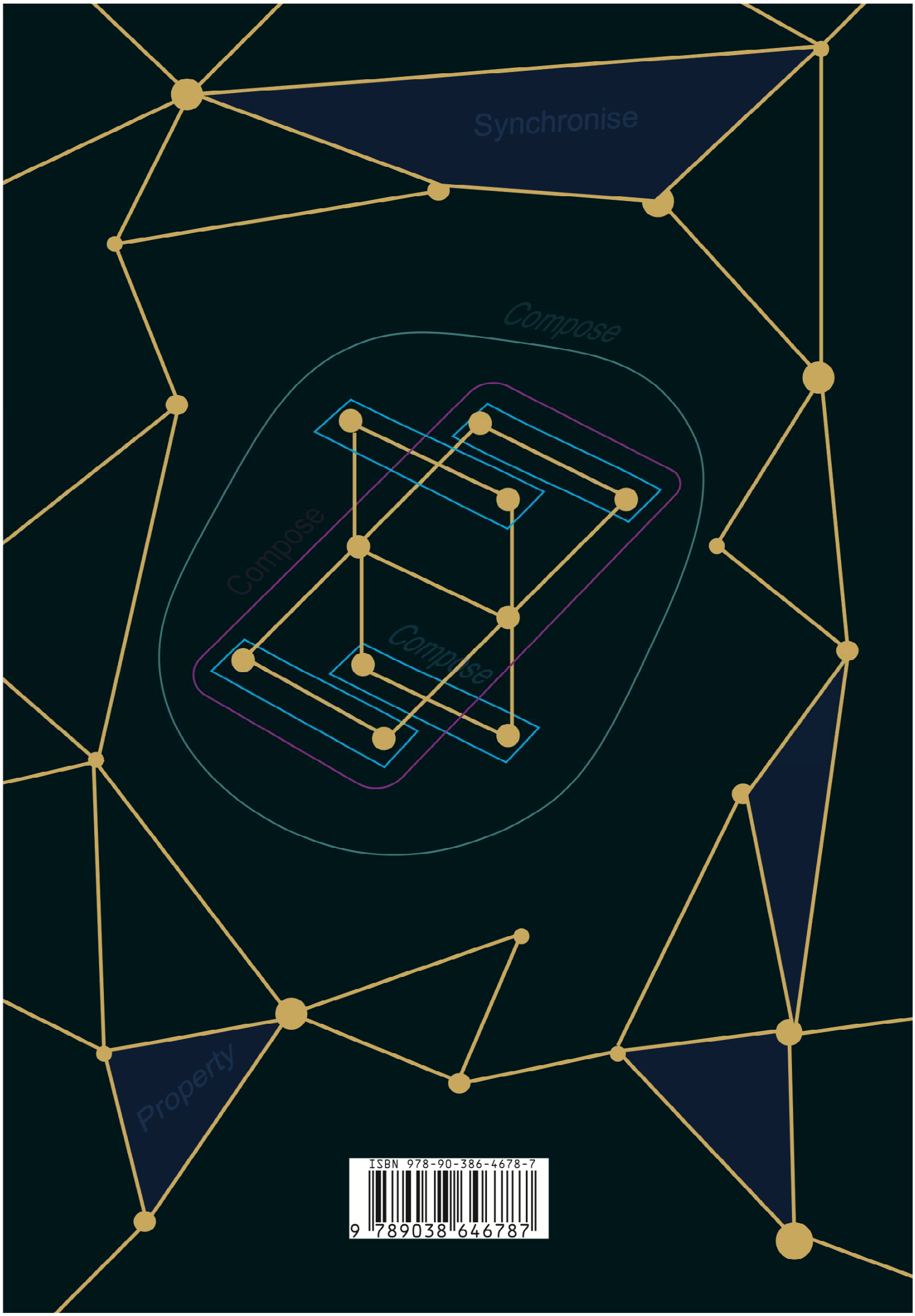
M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations*. Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries*. Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow*. Faculty of Mathematics and Computer Science, TU/e. 2018-21



ISBN 978-90-386-4678-7