EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

On the formal correctness of a model transformation verification technique

de Putter, S.M.J.

Award date:
2014

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

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

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

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer science
Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Assessment committee
dr. A. J. (Anton) Wijs (*tutor*)
dr. R. (Ruurd) Kuiper (*supervisor*)
prof. dr. J. H. (Herman) Geuvers

Section
Software Engineering
and Technology

Date
June 12, 2014

On the formal correctness of a model transformation verification technique

Master's Thesis

S. M. J. (Sander) de Putter

Where innovation starts

Abstract

In Model-driven development (MDD) models are the primary artefact of the software development process. Models reduce the gap between problem domain and implementation. Model transformations are used to manipulate the models (e.g. refactoring, translating, or refining). Using MDD, models can be verified early on to ensure that the software will meet its requirements. Hence, defects (e.g. bugs) in the models can be found early on in the development process. Defects caught early on in the development process are cheaper to fix. Much like software code and models, model transformations can also contain defects. Therefore, model transformations must be verified as well.

Some model transformations, such as refinements, may increase the size of the models they are applied on. The size of a model's state space tends to grow exponentially with the size of the model. Hence, in a sequence of transformations verifying every intermediate model can be time consuming (or even infeasible) as the state space tends to grow exponentially with the size of the model. It is much more efficient to check whether a transformation preserves a given set of properties.

This work formally verifies an existing approach for checking property-preservation for model transformations that may affect synchronising behaviour of parallel processes. The key element of the property-preservation check is checking for preservation of bisimulation. By reasoning about the transformation rules the approach checks whether a transformation is bisimulation preserving for all possible input models. We propose two novel contributions to the bisimulation-preservation check. We show that, without these contributions, transformations are not guaranteed to preserve bisimulation for all input models. Using these notions we show that bisimulation-preservation of transformations can indeed be determined by model checking sets of transformation rules with dependent behaviour.

Acknowledgements

My thanks goes out to my tutor dr. Anton Wijs. We have spend many fruitful hours discussing papers, counter-examples, bisimulation relations and proofs. His remarks and feedback on the project have been invaluable. Furthermore, I would like to thank prof. dr. Herman Geuvers and my (administrative) supervisor dr. Ruurd Kuiper for reviewing this thesis and serving on the examination committee. I would also like to thank prof. dr. Herman Geuvers and drs. Allan van Hulst for helping with the formalizations in Coq. In addition, I would like to thank my comrades Mereke van Garderen, Bram Cappers, Josh Mengerink, Bram van der Sanden for the pleasant and helpful conversations we shared. Special thanks goes out to my family and friends, their support and friendship have been invaluable to me.

Contents

1 Introduction **1**
- 1.1 Terminology . . . 3
- 1.2 Problem description . . . 4
- 1.3 Thesis overview . . . 5

2 Related work **7**
- 2.1 On the correctness of model transformations . . . 7
- 2.2 Model transformation verification approaches . . . 8

3 Transformation of a single process **11**
- 3.1 LTS and LTS equivalence . . . 11
- 3.2 LTS transformation . . . 12
- 3.3 Preservation of branching bisimulation . . . 15

4 Transformation of multiple synchronizing processes **19**
- 4.1 A concurrent system . . . 19
- 4.2 Network transformation . . . 21
- 4.3 Transformation of synchronising behaviour . . . 22
- 4.4 Preservation of branching bisimulation . . . 32

5 Conclusion and Future Work **39**

Appendix **43**

A Coq in a nutshell **43**

B Formalization **45**
- B.1 Transformation of a single process . . . 45
- B.2 Transformation of multiple synchronizing processes . . . 49

Chapter 1

Introduction

In Model-driven development (MDD) models are the primary artefact of the software development process. A simple overview of MDD is presented in Figure 1.1. Models bridge the gap between problem domain and implementation. Using model transformations the models are manipulated, e.g. model transformations can be used to refactor, translate or refine models. When the transformed models are detailed enough, the implementation may be generated (semi-)automatically from the models. Using MDD, models can be verified early on to ensure that the software will meet its requirements. Hence, defects (e.g. bugs) in the models can be found early on in the development process. Defects caught early on in the development process are cheaper to fix [5, 18]. Much like software code and models, model transformations can also contain defects. Therefore, model transformations must be verified and validated as well.

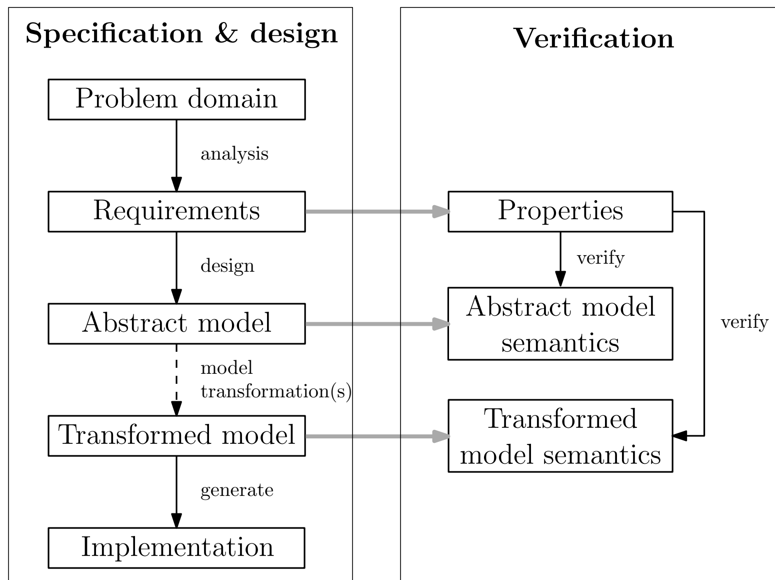


Figure 1.1: Overview of MDD

Many type of models describe dynamic systems where the state of the systems may change over time. For such models requirements can be expressed as *properties* that the model should satisfy such that, with the help of a *model checker*, it can be verified whether the model satisfies the properties [2]. To verify a property of a model the model checker explores the state space implied by the model, which contains all possible states reachable by the system. Model transformations may be applied on large models. Some model transformations, such as refinements, may also increase the size of the models they are applied on. The size of a model's state space tends to grow

exponentially with the size of the model [15]. Therefore, model checking every intermediate model can quickly become infeasible. Instead, it is much more efficient to verify whether a transformation is *property-preserving*. However, the verification of a model transformation is fundamentally more complex than verifying a model [11, 4].

Wijs and Engelen [8, 24, 25] propose an algorithm that checks whether a transformation preserves a given property for any input-model. The proposed algorithm can be used to simplify the process of verifying model transformations. The *property-preservation check* determines whether a transformation preserves a given property by reasoning about the transformation's source- and target-pattern. By relating the source- and target-patterns traditional model-checking techniques [2] can be used to verify whether the transformation preserves a given property.

In order to ensure correct verification results, the property preservation check must be proven correct. This can be done by giving a proof. However, in manual proofs, the steps leading to the proof are often quite large. It is easy to forget details that must be addressed for the correctness of the proof and of the algorithm. A solution is to mechanically verify the proof using (interactive) theorem prover software. Such a theorem prover generates a formal proof using input from the user. Each proof step is verified such that the correctness of the proof can be guaranteed.

The basis of the property-preservation check is the *bisimulation-preservation check*. Two processes are *bisimilar* iff they can simulate each others behaviour. In our previous work [6], we have verified the bisimulation-preservation check for the application of one transformation rule. We provided missing pre-conditions and gave a formal proof of the correctness of the bisimulation-preservation check. In this thesis we continue to verify the model verification approach proposed by Wijs and Engelen. More specifically, we mechanically verify the bisimulation-preservation check for the application of a transformation on a model consisting of concurrent, communicating processes.

Results and contributions. This work formalizes the approach proposed by Wijs and Engelen in an interactive theorem prover. We propose two novel contributions to the theory with the definition of *κ-synchronisation laws* and *cascading rule systems*. We show that, without κ-synchronisation laws and cascading rule systems, transformations are not guaranteed to preserve bisimulation for all input models. Furthermore, we construct a generic relation that relates original and transformed models with respect to the application of a transformation on a model consisting of concurrent, communicating processes. By showing that this relation is a bisimulation relation the *correctness of the bisimulation-preservation check is established*. The proof is *formally verified* using an interactive theorem prover.

The bisimulation-preservation check is limited input models that are admissible with respect to τ-transitions. In addition, the check is limited to rule systems that are confluent, cascading and synchronously uniform. However, it can be checked whether models and rule system satisfy these limitations. Even when a transformation does not preserve bisimulation for a given property, it may still be possible that said property holds for the output model of a specific instance of the transformation. Nevertheless, transformations that do preserve bisimulation can be reused without need for additional verification.

Applications. Once the correctness of the bisimulation-preservation check is formally verified, the output of the bisimulation-preservation check can be trusted. The check can be used to show that a transformation preserves behaviour for all possible input models. Amongst others, the check can be used to verify the correctness of refactoring transformations. Another application is the verification of transformations translating one language into another: if the behaviour of two languages can be expressed as concurrent LTSs, then the check can be used to verify the correctness of the translation transformation. Furthermore, the bisimulation-preservation check can be applied to verify whether a transformation preserves certain properties, i.e. the check allows the model-checking of transformations. Since the check supports abstraction of behaviour, it may be used to check whether refinement transformations are behaviour-preserving (or property-preserving).

1.1 Terminology

Models are simplified or abstract representations of systems. The structure and well-formedness rules of a model is described in a *meta-model*. A *transformation* describes how source models are transformed into corresponding target models according to a set of transformation rules. The *transformation rules* define how constructs in the source language are transformed into constructs in the target language. A *transformation instance* is the process where one or more source models are transformed into one or more target models as specified by the *transformation*. The transformation itself can also conform to a meta-model or transformation language. An overview of how models, meta-models, transformations, and transformation instances are related is shown in Figure 1.2.

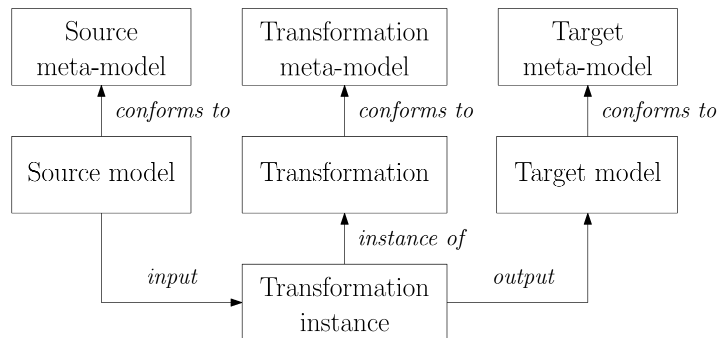


Figure 1.2: Relation between models, meta-models, transformations, and transformation instances

Extensive research has been done on the verification and validation of model transformations. Not all works agree upon the definitions of verification and validation. We follow the IEEE Standard Glossary of Software Engineering [21]:

- *Verification*. "The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase." Informally, verification concerns building the product right. For model transformations, verification entails checking the transformation's properties.

- *Validation*. "The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements." Informally, validation is about building the right product. Therefore, validation of a model transformation considers whether the model transformation adheres to its requirements.

Approaches for the verification of model transformations can be classified in input dependent and input independent approaches:

- *Input independent*. The verification of a transformation's rules. Input independent verification guarantees that a transformation is correct for all possible source models. However, the transformation's rules must be available for this type of verification.

- *Input dependent*. The verification of a transformation's output model given an input model. The correctness of the model transformation transformation is only guaranteed for the instances that have been verified. Since input dependent verification only considers specific transformation instances, these approaches are generally less complex than input independent verification. However, transformations must be re-verified for every new instance.

1.2 Problem description

In MDD models can be analysed to verify that the model satisfies desirable properties. In particular, it is important that safety and liveness properties are satisfied. Informally, a safety property requires that something bad may never happen, e.g. "the model may not enter a deadlock". A liveness property informally requires that something good will eventually happen, e.g. "the model eventually terminates". These kind of properties are expressed over the *dynamic semantics* of a model, i.e. these properties say something about the run-time behaviour of a model. To verify these run-time properties one can use model-checking techniques [2].

As model transformations are software artefacts, like software code or models, they can contain defects. To ensure that a model transformation preserves the dynamic semantics of source models the transformation must be verified. Our work focusses on proving and mechanically verifying the correctness of an *input independent* approach that verifies whether a given model transformation *preserves dynamic semantics.*

The problem. Bisimulation is an equivalence relation which can be used to compare the dynamic semantics of two processes. Two processes are *bisimilar* iff there exists a *bisimulation relation* [2] that shows that both processes can simulate each other's observable behaviour. A model transformation preserves dynamic semantics when a transformation preserves bisimularity between source and target model.

Baldan et al. [3] showed that a transformation is *bisimulation-preserving* if the source- and target-pattern of the rule are bisimular and the communication interface of the process is not changed. However, not only the processes, but also the communication between the processes may be manipulated by model transformations. When a transformation rule changes the communication interface of a process, another process may no longer be able to communicate with the transformed process. It is still possible for a transformation to be bisimulation preserving if the transformation contains other transformation rules which 'update' the affected processes coherently. This idea is incorporated in the model verification approach proposed by Engelen and Wijs [8, 24, 25, 23]. The approach is implemented in the REFINER tool [26]. Nevertheless, no formal proof is given. Our goal is to prove the correctness of the approach proposed by Engelen and Wijs and to guarantee its correctness by means of an interactive theorem prover. For simplicity, we take the theory proposed in [8, 24, 25] as starting point, leaving the extension proposed in [23] as future work.

Our solution. The transformation verification approach proposed by Engelen and Wijs is *input independent* and is suitable for transformations between languages for which the semantics can be expressed in Labelled Transition Systems (LTSs). An *LTS* [2] formally describes what actions a model can take for all reachable states in the state space of a process. Engelen and Wijs present an algorithm, called the *property-preservation check*, which determines whether a transformation preserves certain semantic properties of models it can be applied on. Property preservation is determined by checking for bisimulation preservation after hiding property unrelated actions [16]. The bisimulation-preservation check reasons about the dynamic semantics of models and model transformations using LTSs.

The semantics of a model is formally defined as a network of LTSs. The *LTS network* [12] describes a set of concurrent processes and the synchronisation laws between the processes. The semantics of transformations are formally defined as system of transformation rules, or rule system for short. A *rule system* consists of a set of transformation rules and a set of new synchronisation laws. Each *transformation rule* describes a source LTS-pattern and a target LTS-pattern. A transformation replaces occurrences of the source LTS-pattern by the target LTS-pattern.

To determine bisimulation preservation in cases where the communication interface changes the bisimulation-preservation check must consider synchronisations that are touched directly or indirectly by the transformation, i.e the check must consider combinations of transformation rules. Additionally, the check must consider that it may not always be possible for two actions to syn-

chronise in a given input-model, i.e. the check must consider subsets of transformation rules that touch synchronising behaviour.

The goal of this work is to mechanically verify the bisimulation-preservation check, and to identify the conditions under which the check holds. For verification of proofs we use the Coq proof assistant[1]. Building on the formalizations of our previous work [6], we develop a framework of formalizations aimed towards proving the correctness of the bisimulation-preservation check.

Benefits of the solution. The advantage of using an interactive theorem prover is that the correctness cannot be shown without formalization of all necessary pre-conditions. Hence, no corner-cases can be forgotten. Once the correctness of the bisimulation-preservation check is formally proven, the check can safely be used in the verification of transformations (without further use of the theorem prover).

The bisimulation-preservation check has three benefits. First, traditional model-checking techniques can be used to verify whether a transformation preserves a given property. Second, the output model of the transformation does not have to be re-checked for the preserved properties. Therefore, the verification of models can be done more efficiently. Third, transformations can be re-used, i.e. a transformation does not have to be re-verified for new models with respect to the properties that have been verified for the transformation.

1.3 Thesis overview

In Chapter 2 we review related work. Furthermore, we shortly discuss what is the interpretation of the 'correctness' of model transformations. Chapter 3 discusses bisimulation-preservation with respect to the transformation of a single process LTS. We show that application of a single transformation rule preserves bisimulation if the left and right LTS patterns of the transformation rule, extended with uniquely labelled self-loops, are bisimilar. Many of the notions introduced in Chapter 3 are also applied in the next chapter. Chapter 4 is dedicated to bisimulation-preservation with respect to the transformation of multiple, concurrent process LTSs. We prove that, given proposed pre-conditions, one can determine whether a transformation preserves bisimulation for all possible input-models. In Chapter 5, the conclusion and future work are presented.

Appendix A gives an overview of the Coq interactive theorem prover. Discussed in Appendix B is the formalization of bisimulation-preservation proofs with respect to the transformation of multiple, concurrent process LTSs.

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

Chapter 2

Related work

Numerous approaches for verification of model transformations have been proposed in the literature. Yet, research on the verification of the transformation mechanism is still in its infancy [1]. Most related works discuss approaches to verify instances of model transformations. This work considers the more general case: the verification of an approach for verifying model transformations including the transformation mechanism. Since there is little work on the verification of transformation verification approaches we will discuss transformation verification approaches, not limiting ourselves to the verification of these approaches.

Most input dependent approaches do not require verification of the approach itself since the output is mostly re-verified or compared with the input model. More related to our work are the input independent approaches, for which a proof is required showing that verification of the transformation rules is sufficient to identify preservation of a given property. Therefore, we will only discuss *input independent* approaches.

First, we discuss a few notions of correctness for model transformation. Next, we discuss alternative and complementary approaches and compare this work to ours. We distinguish between the two most common techniques: theorem proving and model checking.

2.1 On the correctness of model transformations

In the literature the 'correctness' of model transformations has many interpretations. Not all interpretations necessarily guarantee full preservation of behaviour. We will distinguish between the *notions of correctness* proposed by Rahim and Whittle [1]:

- *Type-correctness.* A transformation is type-correct if the target model conforms to the abstract syntax of the output language, i.e. elements of the target model are instances of elements of the target meta-model.

- *Preservation of static semantics.* A transformation preserves static semantics if the output model of the transformation is well-formed with respect to the target meta-model. If output models are well-formed, then they are also type-correct.

- *Preservation of dynamic semantics.* A transformation preserves dynamic semantics when the output model preserves a given property of the source model, such as safety properties, liveness properties, and properties relating the semantics of the output model to the semantics of the source model.

- *Correspondence between source and target.* According to this notion a transformation is correct when the elements of the output model correspond to elements of the source model; e.g. the transformation of a UML class diagram to a relation database proposed by Ehrig et al. [7] is considered correct because the transformation generates an entity for each class in the class diagram.

- *Semantics of model transformation.* This notion considers properties of the transformation itself, such as termination and confluence.

2.2 Model transformation verification approaches

Theorem proving. Stenzel et al. [20] propose a calculus for operational Query View Transform (QVT) transformations. The calculus has one rule for every operational QVT expression. The correctness of these rules are proven using the KIV interactive theorem prover. The rules express under which preconditions a QVT expression maintains a given property. The approach can be used to determine *type-correctness* and *preservation of dynamic semantics*. The authors apply their approach to verify security-properties for a transformation from UML class diagrams to Java code. KIV is used to reason about the formal semantics of the source- and target-languages. A disadvantage of the approach is that KIV must also be used to reason about the properties of transformations. According to the authors: "Interactive theorem provers require quite a lot of experience to use successfully, because the user must know the logic, the input language, how things are formalized, and how to utilize the strength of the tool." Therefore, verification of transformations with this approach requires high efforts or experience in theorem proving.

A model transformation verification approach is presented by Giese et al. [10] where model transformations are specified using *Triple Graph Grammars* (TGGs). The model transformation is defined by a set of transformation rules. Each transformation rule consists of a source, target and correspondence graph which are defined in terms of the meta-models of the respective languages. The correspondence model defines the mapping between the source and target graphs. *Preservation of dynamic semantics* is proven by showing that bisimulation is a congruence with respect to the specified transformation rules. The authors use an interactive theorem prover to guarantee the correctness of the proof. Changing or adding transformation rules requires a new proof and re-verification, again requiring the investment of valuable resources.

Poernomo and Terrel [17] extend the *proof-as-model-transformation* approach by supporting partially ordered model transformation specifications. Partially ordered traversal mappings are defined between classes of the source and target meta-models. A transformation is specified in Coq as a formula of the form $\forall M_{src} \in T_{src}.Pre(M_{src}) \implies \exists M_{trgt} \in T_{trgt}.Post(M_{src}, M_{trgt})$, where $Pre(M_{src})$ is a precondition over instance M_{src} of meta-model T_{src}, and $Post(M_{src}, M_{trgt})$ is a post-condition describing the required relation between instances M_{src} and M_{trgt} of meta-models T_{src} and T_{trgt} respectively. The authors apply their approach to a simple transformation indicating *correspondence between source and target* and *type-correctness*. Since the transformation is extracted from the proof the transformation is correct by construction. However, the proof for this simple transformation is considerable and requires manual input. It is likely that an order of magnitude more manual input is needed to guide the proof assistant as the proof's search space exponentially increases with complexity.

A subset of UML, called UML Reactive System Development Support (UML-RSDS), is used by Lano and Kolahdouz-Rahimi [13] to specify and verify model transformations. In UML-RSDS model transformations are specified declaratively using OCL constraints or by using operations of meta-model classes. A set of transformation rules is represented by a UML class. The rule-set has an application policy which controls the order and conditions of the applications of its rules. Transformation rules are specified as operations using pre- and postcondition pairs in the OCL notation. The precondition of the rule specifies which elements of the source-meta-model are considered by the rule and when the rule is applicable. The postcondition of the rule specifies how elements of the target meta-model are related to the considered elements of the source meta-model. For verification, the transformation is automatically translated to the B formalism. Using B machines and inference rules, *type-correctness* and *preservation of static semantics* can be checked. The disadvantage of the approach is that it only verifies the specification of model transformations. Further verification is required to verify whether a transformation's implementation matches its specification.

In summary, there is little work on the verification of (input independent) transformations with

respect to dynamic semantics [1]. The work that considers dynamic semantics is often not fully automated. Our contribution in this area is the correctness proof of a fully automatic approach for the verification of model transformations. For more complete verification our work can be used together with other approaches that verify preservation of static semantics, termination, and confluence. Comparing our work with these other approaches we observe that most approaches require additional use of theorem proving to validate the model transformation. In our work we prove the correctness of a transformation verification approach that uses basic model-checking principles. This approach requires no knowledge of theorem proving and is more accessible to those with little experience in theorem proving.

Model checking. Garcia and Möller [9] propose an approach translating transformation rules to the $^{+}$CAL model-checking language. The EMOF meta-models and OCL statements are automatically translated to $^{+}$CAL. The transformation is expressed as a $^{+}$CAL algorithm operating on Abstract Syntax Trees. Additionally, the transformation is annotated with assumptions about its input and assertions about the system state. The $^{+}$CAL specification is then checked for *preservation of static semantics* and *termination*. It is not clear whether the translation from EMOF meta-models and OCL to $^{+}$CAL is verified as well.

A transformation verification approach proposed by Baldan et al. [3] verifies transformation rules for Open Nets (ONets). Open Nets are a reactive extension of Petri Nets, allowing the specification of processes that communicate with unknown processes via an interface. The authors show that a transformation rule is weak-bisimulation-preserving iff the left- and right-patterns are weakly bisimilar. The interface to external processes must remain untouched. The approach is *dynamic-semantics-preserving*. The proposed approach is limited to the verification of transformations that leave the communication interface intact. Furthermore, to determine semantic preservation the authors use weak bisimulation which does not respect the branching structure of a process. Therefore, weak bisimulation is not adequate with properties expressing inevitable reachability [22].

Lúcio and Vangheluwe [14] use symbolic execution to verify model transformations written in the DSLTrans transformation language. The approach reasons about the meta-models and the structure of the transformation. In DSLTrans transformation rules are encapsulated in layers. These layers are connected with each other forming a graph. Because the transformations are structured using these layers the model-checker can build a state space consisting of all possible execution combinations of transformation rules. A property meta-model is constructed from the source- and target-meta-models of the transformation. Therefore, properties expressing *correspondence between source and target* can be formulated and verified. Furthermore, DSLTrans transformations are *terminating* and *confluent* by construction. The DSLTrans language is not Turing complete. Hence, some transformations may not be expressible in DSLTrans. Furthermore, the language in which properties are expressed is in a relatively basic state. Properties about attributes cannot be expressed. These attributes should also be addressed symbolically, requiring an extension of the algorithm.

In summary, again not many approaches consider correctness with respect to dynamic semantics. The work that does consider dynamic semantics only allows transformations which do not change the communication interface of a process. In contrast, the approach we verify can also determine preservation of dynamic semantics when the communication interface changes. If communicating parties are all transformed coherently the transformation may still preserve dynamic semantics. We prove the correctness of the model vertification approach proposed by Engelen and Wijs with respect to branching bisimulation. Unlike weak bisimulation, branching bisimulation respects the branching structure of a process. Unlike weak bisimulation, branching bisimulation is adequate with properties expressing inevitable reachability [22]. In contrast with other work, we mechanically-verify the transformation verification approach. This guarantees that the approach is correct and bug-free.

Chapter 3

Transformation of a single process

In this chapter first the notion of LTS and branching bisimulation equivalence are discussed in Section 3.1. Next, in Section 3.2 we discuss the transformation of an LTS given a transformation rule and its matches. Finally, in Section 3.3, it is shown that one can verify whether a transformation rule is bisimulation-preserving for all possible input processes. The most significant formalizations are given in Section B.1 of Appendix B.

3.1 LTS and LTS equivalence

Labelled Transition System. The semantics of processes and transformation rules are expressed in terms of *Labelled Transition Systems* (LTSs). The LTS is formally defined in Definition 1. For a process an LTS describes all possible behaviour of the process. For transformation rules, LTSs explain how a given LTS is refined: occurrences of one pattern LTS are replaced by another pattern LTS. The traditional definition of LTS [2] contains a single initial state. In contrast, an LTS as defined by Definition 1 contains a non-empty set of initial states. This deviation makes it possible to describe both traditional LTSs and pattern LTSs.

Definition 1 (Labelled Transition System)**.** *A Labelled Transition System $\mathcal{G}$ is a four-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 (including the invisible action τ);*
- $\mathcal{T}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \times \mathcal{S}_\mathcal{G}$ *a transition relation;*
- $\mathcal{I}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G}$ *a non-empty set of initial states*

Action labels will be denoted by a, b, c, etc. 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 reflexive-transitive closure of the binary relation $\xrightarrow{\tau}_\mathcal{G}$ is denoted by $\Rightarrow_\mathcal{G}$. Furthermore, the intersection of two LTSs $\mathcal{G}$ and $\mathcal{H}$ is defined as $\mathcal{G} \cap \mathcal{H} = (\mathcal{S}_\mathcal{G} \cap \mathcal{S}_\mathcal{H}, \mathcal{A}_\mathcal{G} \cap \mathcal{A}_\mathcal{H}, \mathcal{T}_\mathcal{G} \cap \mathcal{T}_\mathcal{H}, \mathcal{I}_\mathcal{G} \cap \mathcal{I}_\mathcal{H})$ [8]. An example of an LTS is shown in Figure 3.1. States are indicated as circles. Initial states are coloured black. The labelled arrows between states indicate transitions.

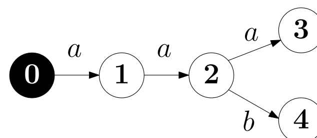


Figure 3.1: Example of an LTS

LTS equivalence. The *branching bisimulation* [2] equivalence relation is used to compare LTSs. Branching bisimulation is sensitive to the branching structure of an LTS and supports abstraction from actions. Furthermore, branching bisimulation is adequate with properties expressing inevitable reachability [22]. Therefore, branching bisimulation preserves safety and liveness properties. Branching bisimulation is defined as follows:

Definition 2 (Branching bisimulation). *A binary relation B between two LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$ is a branching bisimulation iff the following holds:*
For all $s \in \mathcal{S}_{\mathcal{G}_1}$ and $t \in \mathcal{S}_{\mathcal{G}_2}$, $s\ B\ t$ implies:

1. *if $s \xrightarrow{a}_{\mathcal{G}_1} s'$ then*
   - *either $a = \tau$ with $s'\ B\ t$;*
   - *or $t \Rightarrow_{\mathcal{G}_2} \hat{t} \xrightarrow{a}_{\mathcal{G}_2} t'$ with $s\ B\ \hat{t}$ and $s'\ B\ t'$.*
2. *the symmetric case: if $t \xrightarrow{a}_{\mathcal{G}_2} t'$ then*
   - *either $a = \tau$ with $s\ B\ t'$;*
   - *or $s \Rightarrow_{\mathcal{G}_1} \hat{s} \xrightarrow{a}_{\mathcal{G}_1} s'$ with $\hat{s}\ B\ t$ and $s'\ B\ t'$.*

The two conditions, called the *transfer conditions*, enforce branching bisimilarity between states as depicted in Figure 3.2, the dashed lines indicate related states. Two states s and t are *branching bisimilar*, denoted $s \underline{\leftrightarrow}_b t$, if there is a branching bisimulation B with $s\ B\ t$. Furthermore, two sets of states $S_1 \subseteq \mathcal{S}_{\mathcal{G}_1}, S_2 \subseteq \mathcal{S}_{\mathcal{G}_2}$ are called branching bisimilar, denoted $S_1 \underline{\leftrightarrow}_b S_2$, iff $\forall s_1 \in S_1 : (\exists s_2 \in S_2 : s_1 \underline{\leftrightarrow}_b s_2)$ and vice versa. We say two LTSs $\mathcal{G}_1, \mathcal{G}_2$ are branching bisimilar, denoted as $\mathcal{G}_1 \underline{\leftrightarrow}_b \mathcal{G}_2$, iff $\mathcal{I}_{\mathcal{G}_1} \underline{\leftrightarrow}_b \mathcal{I}_{\mathcal{G}_2}$.

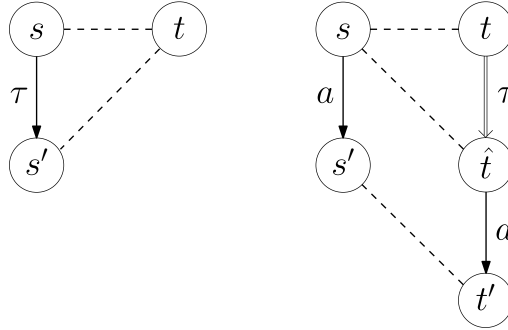


Figure 3.2: Visualization of the branching condition of branching bisimulation

3.2 LTS transformation

A transformation rule and its match. Refinement of an LTS is achieved by means of an LTS transformation. Such an LTS transformation is defined through a transformation rule. When a transformation rule has a match on the given LTS the transformation can be applied. Definition 3 formally defines a transformation rule.

Definition 3 (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 $\mathcal{I}_\mathcal{L} = \mathcal{I}_\mathcal{R} = (\mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R})$.*

The initial states of the left and right pattern LTSs are known as the *glue-states* and consist of the set of states that are part of both pattern LTSs, i.e. $\mathcal{I}_\mathcal{L} = \mathcal{I}_\mathcal{R} = (\mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R})$. When applying a

transformation rule to an LTS, the changes are applied relative to these glue-states. An illustration of a transformation rule is given in Figure 3.3. Glue-states are coloured black, and non-glue-states are coloured white. The open arrow emphasizes that the left pattern LTS is replaced by the right pattern LTS. The transformation rule replaces two sequential a-actions with two sequential a'-actions while replacing the state connecting the two transitions. The intersection of the two pattern LTSs is shown on the right side of the figure.

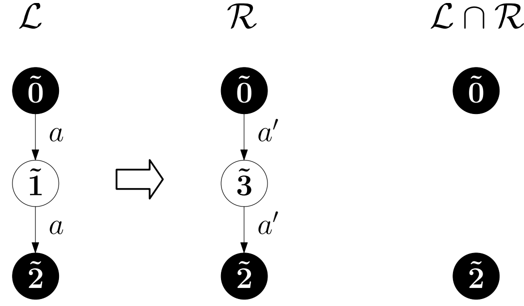


Figure 3.3: Illustration of a transformation rule r

A transformation rule $r = (\mathcal{L}, \mathcal{R})$ is *applicable* on a given LTS $\mathcal{G}$ iff a *match* $m : \mathcal{S}_\mathcal{L} \to \mathcal{S}_\mathcal{G}$ on $\mathcal{G}$ exists according to Definition 4.

Definition 4 (Match)**.** *An pattern LTS $\mathcal{H} = (\mathcal{S}_\mathcal{H}, \mathcal{S}_\mathcal{H}, \mathcal{T}_\mathcal{H}, \mathcal{I}_\mathcal{H})$ has a match $m : \mathcal{S}_\mathcal{H} \to \mathcal{S}_\mathcal{G}$ on an LTS $\mathcal{G} = (\mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G})$ iff the following holds:*

1. *m is injective;*
2. $\forall i \in \mathcal{I}_\mathcal{G}, s \in \mathcal{S}_\mathcal{H} : m(s) = i \implies s \in \mathcal{I}_\mathcal{H}$;
3. $\forall s_1 \xrightarrow{a}_\mathcal{H} s_2 : m(s_1) \xrightarrow{a}_\mathcal{G} m(s_2)$;
4. $\forall s \in \mathcal{S}_\mathcal{H} \setminus \mathcal{I}_\mathcal{H}, p \in \mathcal{S}_\mathcal{G}$:

   (D1) $m(s) \xrightarrow{a}_\mathcal{G} p \implies (\exists s' \in \mathcal{S}_\mathcal{H} : s \xrightarrow{a}_\mathcal{H} s' \wedge m(s') = p)$;

   (D2) $p \xrightarrow{a}_\mathcal{G} m(s) \implies (\exists s' \in \mathcal{S}_\mathcal{H} : s' \xrightarrow{a}_\mathcal{H} s \wedge m(s') = p)$.

A match is an embedding of pattern LTS $\mathcal{H}$ in LTS $\mathcal{G}$. The four conditions in Definition 4 are known as the *glueing conditions*. A match is only valid when these glueing conditions hold. The *second condition* ensures that the initial states of an LTS are not removed by a transformation with match m, i.e. if a state $s \in \mathcal{S}_\mathcal{H}$ matches on an initial state $i \in \mathcal{I}_\mathcal{G}$ then s must be a glue-state. The *third condition* of Definition 4 enforces that for every transition in $\mathcal{S}_\mathcal{H}$ there is a matching transition in $\mathcal{G}$ such that the labels are equal and its states are matched in the same direction. The *fourth condition* expresses the dangling conditions. The *dangling conditions* enforce the existence of a transition from or to the match of a non-glue-state. As a consequence, the dangling conditions rule out removal of transitions that are not explicitly present in $\mathcal{H}$. Therefore, the transformation of Figure 3.3 cannot be applied on LTS $\mathcal{G}$ in Figure 3.4. For dangling condition $(D2)$ the direction of the a-transition is simply reversed. The set $m(\mathcal{S}) = \{m(s) \in \mathcal{G} \mid s \in \mathcal{S}\}$ is the image of a set of states $\mathcal{S}$ through match m on an LTS $\mathcal{G}$.

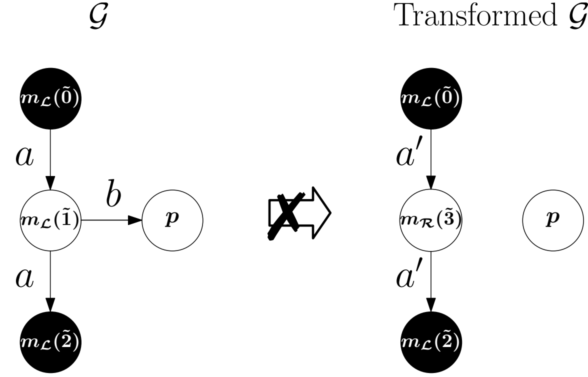


Figure 3.4: Dangling condition ($D1$) invalidates this match, preventing the removal of the b-transition

Lemma 1 describes that whenever a transition is not matched but a state of the transition is matched, then the matched state must be a glue-state.

Lemma 1. *Let $\mathcal{H} = (\mathcal{S}_\mathcal{H}, \mathcal{S}_\mathcal{H}, \mathcal{T}_\mathcal{H}, \mathcal{I}_\mathcal{H})$ be an pattern LTS with match $m : \mathcal{S}_\mathcal{H} \to \mathcal{S}_\mathcal{G}$ on an LTS $\mathcal{G} = (\mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G})$, then*

$$\begin{aligned}&\forall s_1, s_2 \in \mathcal{S}_\mathcal{G} :\ s_1 \xrightarrow{a}_\mathcal{G} s_2 \implies \\ &\qquad \forall p \in \mathcal{S}_\mathcal{H} : \begin{pmatrix} & (m(p) = s_1 \land \neg(\exists p_2 \in \mathcal{S}_\mathcal{H} : m(p_2) = s_2 \land p \xrightarrow{a}_\mathcal{H} p_2)) \\ \lor & (m(p) = s_2 \land \neg(\exists p_1 \in \mathcal{S}_\mathcal{H} : m(p_1) = s_1 \land p_1 \xrightarrow{a}_\mathcal{H} p)) \end{pmatrix} \implies p \in \mathcal{I}_\mathcal{H}\end{aligned}$$

Proof. Let $s_1, s_2 \in \mathcal{S}_\mathcal{G}$ such that $s_1 \xrightarrow{a}_\mathcal{G} s_2$. Assume for a contradiction that $p \notin \mathcal{I}_\mathcal{H}$. We distinguish two cases:

- Case: $m(p) = s_1 \land \neg(\exists p_2 \in \mathcal{S}_\mathcal{H} : m(p_2) = s_2 \land p \xrightarrow{a}_\mathcal{H} p_2)$. By (D1) of Definition 4 there exists a $p_2 \in \mathcal{S}_\mathcal{H}$ such that $m(p_2) = s_2$ and $p \xrightarrow{a}_\mathcal{H} p_2$. Hence, we have a contradiction.
- Case: $m(p) = s_2 \land \neg(\exists p_1 \in \mathcal{S}_\mathcal{H} : m(p_1) = s_1 \land p_1 \xrightarrow{a}_\mathcal{H} p)$. By (D2) of Definition 4 there exists a $p_1 \in \mathcal{S}_\mathcal{H}$ such that $m(p_1) = s_1$ and $p_1 \xrightarrow{a}_\mathcal{H} p$. Again, we have a contradiction.

□

Transformation of an LTS. A transformation is implemented using the *double-pushout* (DPO) approach [19]. Hence, a transformation rule can only be applied if there exists a match such that the glueing conditions hold. Definition 5 defines the transformation of an LTS $\mathcal{G}$ by means of an applicable transformation rule r.

Definition 5 (LTS Transformation). *Let $\mathcal{G} = (\mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G})$ be an LTS, let $r = (\mathcal{L}, \mathcal{R})$ be a transformation rule, and let $T(\mathcal{G})$ denote the LTS resulting from the transformation of LTS $\mathcal{G}$. Furthermore, let $m : \mathcal{S}_\mathcal{L} \to \mathcal{S}_\mathcal{G}$ and $\hat{m} : \mathcal{S}_\mathcal{R} \to \mathcal{S}_{T(\mathcal{G})}$ be the matches of rule r such that $\hat{m}(s) = m(s)$ for all states $s \in \mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R}$. The transformation of LTS $\mathcal{G}$, via rule r with matches m and $\hat{m}$, is defined as $T(\mathcal{G}) = (\mathcal{S}_m, \mathcal{A}_m, \mathcal{T}_m, \mathcal{I}_\mathcal{G})$ where*

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

The set of states $\mathcal{S}_m$ obtained after transformation consists of the states in $\mathcal{S}_\mathcal{G}$ without the states to be removed (matches of states in $\mathcal{S}_\mathcal{L} \setminus \mathcal{S}_\mathcal{R}$), and with states to be added (states in $\mathcal{S}_\mathcal{R} \setminus \mathcal{S}_\mathcal{L}$). The added states must be 'fresh' with respect to the states in $\mathcal{S}_\mathcal{G}$, i.e. we must have $\forall s \in \mathcal{S}_\mathcal{R} \setminus \mathcal{S}_\mathcal{L} : \hat{m}(s) \notin \mathcal{S}_\mathcal{G}$. The new transitions $\mathcal{T}_m$ consist of the transitions $\mathcal{T}_\mathcal{G}$ where matched transitions from $\mathcal{T}_\mathcal{L}$ are removed and matched transitions from $\mathcal{T}_\mathcal{R}$ are added. Since we have $\hat{m}(s) = m(s)$ for all glue-states $s \in \mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R}$, the transitions matched by both $\mathcal{T}_\mathcal{L}$ and $\mathcal{T}_\mathcal{R}$ are left intact.

Consider the transformation rule r shown previously in Figure 3.3. An application of r on an LTS is shown in Figure 3.5. The $\Rrightarrow$ symbol indicates application of a transformation rule. In this example, states $\tilde{0}$, $\tilde{1}$, and $\tilde{2}$ of the transformation rule, are matched to states 0, 1, and 2 of $\mathcal{G}$ respectively. The left-pattern of r does not match on states 1, 2, and 3 as this would remove the b-transition and violate the dangling conditions of a match. The right most LTS shows the intersection between $\mathcal{G}$ and $T(\mathcal{G})$ and shows what part of $\mathcal{G}$ is left unchanged by the transformation.

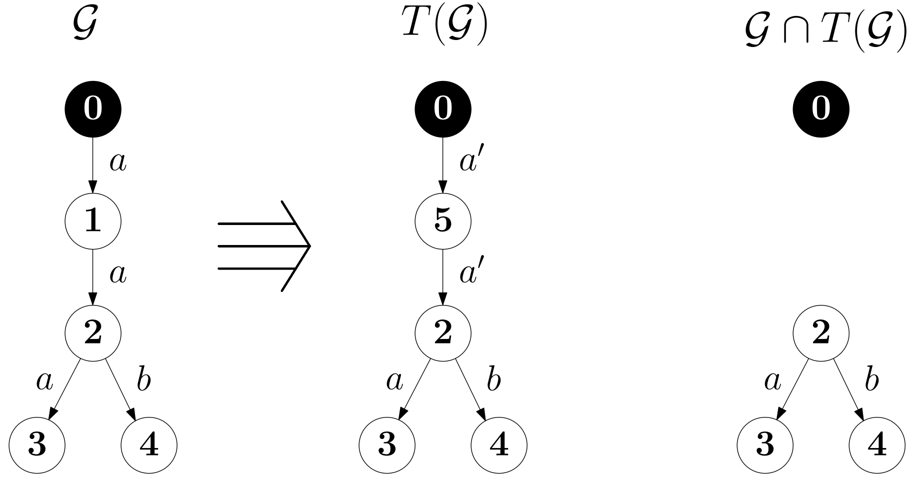


Figure 3.5: Example of an LTS transformation

3.3 Preservation of branching bisimulation

Handling incoming and outgoing transitions for pattern LTSs. Glue-states form the interface between the source and target LTS of a transformation. Matches of glue-states of a transformation rule r may have outgoing and incoming transitions that are not present in the patterns of r. To make these possible transitions explicit κ-self-loops are added to the glue-states of transformation rule r. An LTS extended with self-loops on the glue states is formalized in Definition 6. The κ-extended transformation rule can now be defined as $r^\kappa = (\mathcal{L}^\kappa, \mathcal{R}^\kappa)$. The self-loop of a glue-state s is labelled with a unique label $\kappa_s \notin \mathcal{A}_\mathcal{L} \cup \mathcal{A}_\mathcal{R}$.

Definition 6 (LTS extended with κ-self-loops)**.** *The LTS $\mathcal{H}$ extended with κ-self-loops is defined as follows:*

$$\mathcal{H}^\kappa = (\mathcal{S}_\mathcal{H}, \mathcal{A}_\mathcal{H} \cup \{\kappa_s \mid s \in \mathcal{I}_\mathcal{H}\}, \mathcal{T}_\mathcal{H} \cup \{(s, \kappa_s, s) \mid s \in \mathcal{I}_\mathcal{H}\}, \mathcal{I}_\mathcal{H}).$$

The κ-self-loops ensure that glue-states are at least related to themselves. This prevents bisimilarity of two different glue-states which are matched on states with different incoming and outgoing transitions where those transitions are not explicitly present in the patterns of the transformation rule. Such a situation is illustrated in Figure 3.6. Possible outgoing and incoming transitions that are not present in the transformation rule are represented by the κ-selfloops.

Because κ-actions are not τ-actions, it is clear that any sequence of τ-transition in a κ-extended LTS $\mathcal{H}^\kappa$ must be present in the original LTS $\mathcal{H}$ as well. This intuition is formalized in Lemma 2.

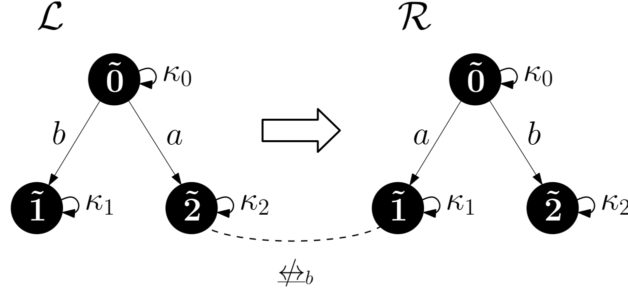


Figure 3.6: The κ-self-loops ensure $2 \not\leftrightarrow_b 1$

Lemma 2. *Let $\mathcal{H}$ be an LTS, then*

$$\forall s, s' \in \mathcal{S}_{\mathcal{H}} : (s \Rightarrow_{\mathcal{H}^\kappa} s') \implies (s \Rightarrow_{\mathcal{H}} s')$$

Proof. By structural induction on $\Rightarrow_{\mathcal{H}^\kappa}$:

- Base case: $s \Rightarrow_{\mathcal{H}^\kappa} s$. We have to show $s \Rightarrow_{\mathcal{H}} s$, this follows directly from the reflexivity of $\Rightarrow$.
- Base case: $s \xrightarrow{\tau}_{\mathcal{H}^\kappa} s'$. We have to show $s \Rightarrow_{\mathcal{H}} s'$.

  Since $s \xrightarrow{\tau}_{\mathcal{H}^\kappa} s'$ and since the κ-extension only adds self loops, we have $s \xrightarrow{\tau}_{\mathcal{H}} s'$ or $s = s' \wedge \tau = \kappa_s$. Since $\kappa_s \neq \tau$ the latter case cannot occur. Hence, we have $s \xrightarrow{\tau}_{\mathcal{H}} s'$. By definition of $\Rightarrow$ it follows that $s \Rightarrow_{\mathcal{H}} s'$.
- Step case: $s \Rightarrow_{\mathcal{H}^\kappa} \hat{s}$ and $\hat{s} \Rightarrow_{\mathcal{H}^\kappa} s'$ with Induction Hypothesis: $(s \Rightarrow_{\mathcal{H}^\kappa} \hat{s}) \implies (s \Rightarrow_{\mathcal{H}} \hat{s})$ and $(\hat{s} \Rightarrow_{\mathcal{H}^\kappa} s') \implies (\hat{s} \Rightarrow_{\mathcal{H}} s')$. We have to show $s \Rightarrow_{\mathcal{H}} s'$.

  By the Induction Hypothesis we have $s \Rightarrow_{\mathcal{H}} \hat{s}$ and $\hat{s} \Rightarrow_{\mathcal{H}} s'$. The proof follows from the transitivity of $\Rightarrow$.

□

In a branching bisimulation relation between the two patterns of a transformation rule extended with κ-self-loops, a glue-state s must be related to itself, since it is the only state having a κ_s-transition. Hence, a state $s' \in \mathcal{S}_{\mathcal{R}}$ can only be related to a glue-state $s \in \mathcal{I}_{\mathcal{L}}$ if there exists a τ-path from s' to s, where the τ-path may have length zero ($s = s'$). This idea is expressed in Lemma 3.

Lemma 3. *Let B be a branching bisimulation relation such that $\mathcal{L}^\kappa \leftrightarrow_b \mathcal{R}^\kappa$, then*

$$\forall s \in \mathcal{I}_{\mathcal{L}}, s' \in \mathcal{S}_{\mathcal{R}} : s\ B\ s' \implies (s' \Rightarrow_{\mathcal{R}} s)$$

Proof. We have a branching bisimulation relation B such that $\mathcal{L}^\kappa \leftrightarrow_b \mathcal{R}^\kappa$. Let state $s \in \mathcal{I}_{\mathcal{L}}$ and state $s' \in \mathcal{S}_{\mathcal{R}}$. Given $s\ B\ s'$, we have to show that $s' \Rightarrow_{\mathcal{R}} s$. Since $s \in \mathcal{I}_{\mathcal{L}}$, we have $s \xrightarrow{\kappa_s}_{\mathcal{R}} s$ by Definition 6. We distinguish two cases corresponding to the branching condition of branching bisimulation:

- Case $\kappa_s = \tau$ with sBs':

  By Definition 6 we have $\kappa_s \neq \tau$. This contradicts the case's assumption. Therefore, this case cannot occur.
- Case $\exists \hat{t}, t' \in \mathcal{S}_{\mathcal{R}^\kappa} : s' \Rightarrow_{\mathcal{R}^\kappa} \hat{t} \xrightarrow{\kappa_s}_{\mathcal{R}^\kappa} t'$ with $s\ B\ \hat{t}$ and $s\ B\ t'$:

  The κ_s-transition only occurs as $s \xrightarrow{\kappa_s}_{\mathcal{R}^\kappa} s$, therefore, we must have $\hat{t} = s$ and $t' = s$. Hence, we have $s' \Rightarrow_{\mathcal{R}^\kappa} s$. By Lemma 2, the proof follows.

□

The bisimulation-preservation check. A transformation is branching-bisimulation-preserving if the patterns of a transformation rule extended with κ-self-loops are branching bisimilar. This is expressed in Proposition 1. The bisimulation-preservation check follows directly from the proposition.

Proposition 1. *Let $\mathcal{G}$ be an LTS, let r be a transformation rule with matches $m : \mathcal{S}_{\mathcal{L}} \to \mathcal{S}_{\mathcal{G}}$ and $\hat{m} : \mathcal{S}_{\mathcal{L}} \to \mathcal{S}_{T(\mathcal{G})}$ with $m(s) = \hat{m}(s)$ for all $s \in \mathcal{S}_{\mathcal{L}} \cap \mathcal{S}_{\mathcal{R}}$. Then, the following holds:*

$$\mathcal{L}^{\kappa} \leftrightarrow_b \mathcal{R}^{\kappa} \implies \mathcal{G} \leftrightarrow_b T(\mathcal{G})$$

Proof. By definition of bisimilarity between LTSs, we have $\mathcal{G} \leftrightarrow_b T(\mathcal{G})$ iff there exists a branching bisimulation relation C such that $\mathcal{I}_{\mathcal{G}}\ C\ \mathcal{I}_{T(\mathcal{G})}$. Before constructing relation C, we first define a relation D which relates all states of LTS $\mathcal{G}$ not removed by the transformation to themselves, we have $D = \{(s, s) \mid s \in \mathcal{S}_{\mathcal{G}} \setminus m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})\}$. Furthermore, since $\mathcal{L}^{\kappa} \leftrightarrow_b \mathcal{R}^{\kappa}$, there also exists a branching bisimulation relation B with $\mathcal{I}_{\mathcal{L}}\ B\ \mathcal{I}_{\mathcal{R}}$ with $\{(s, s) \mid s \in \mathcal{I}_{\mathcal{L}}\} \subseteq B$. To relate states that are removed from $\mathcal{G}$ and states that are added to $\mathcal{G}$ we define relation $\hat{B} = \{(m(q), \hat{m}(p) \mid q\ B\ p\}$. Relation C is constructed as follows: $C = D \cup \hat{B}$. Lemma 4 shows that a state $\hat{m}(p) \in \mathcal{S}_{\mathcal{R}}$ simulates the behaviour of a state $m(q) \in \mathcal{S}_{\mathcal{L}}$ if $q\ B\ p$. We will present Lemma 4 after the proof of Proposition 1. We now prove that C is a branching bisimulation relation, i.e. we show that Definition 2 holds for C.

- C relates the initial states of $\mathcal{G}$ and $T(\mathcal{G})$, i.e. $\mathcal{I}_{\mathcal{G}}\ C\ \mathcal{I}_{T(\mathcal{G})}$.

  Since we have $\mathcal{I}_{\mathcal{G}} = \mathcal{I}_{T(\mathcal{G})}$ we only have to show $\forall s \in \mathcal{I}_{\mathcal{G}} : \exists t \in \mathcal{I}_{\mathcal{G}} : s\ C\ t$. Let s be an arbitrary state in $\mathcal{I}_{\mathcal{G}}$, we can instantiate the existential quantifier with s. What remains to be shown is: $s\ C\ s$. Since the second condition of Definition 4 prevents the removal of initial states, i.e. $s \notin m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$, we have $s\ D\ s$. Hence, by definition, we have $s\ C\ s$.

- If $s\ C\ t$ and $s \xrightarrow{a}_{\mathcal{G}} s'$ then either $a = \tau \wedge s'\ C\ t$, or $t \Rightarrow_{T(\mathcal{G})} \hat{t} \xrightarrow{a}_{T(\mathcal{G})} t' \wedge s\ C\ \hat{t} \wedge s'\ C\ t'$.

  We distinguish two cases: $s\ D\ t$ and $s\ \hat{B}\ t$.

  1. Case $s\ D\ t$. We have $s = t$, $s \in \mathcal{S}_{\mathcal{G}} \setminus m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$, and $s \xrightarrow{a}_{\mathcal{G}} s'$.
     Again, we distinguish two cases:
     (a) Case $s' \in \mathcal{S}_{\mathcal{G}} \setminus m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$. Both s and s' are not removed by the transformation. We distinguish two cases:
        - Case $m(q) = s \wedge m(q') = s' \wedge q \xrightarrow{a}_{\mathcal{L}} q'$ (there exists a transition in $\mathcal{T}_{\mathcal{L}}$ matching $s \xrightarrow{a}_{\mathcal{G}} s'$). Then $q \in \mathcal{I}_{\mathcal{L}}$ and $q' \in \mathcal{I}_{\mathcal{L}}$. Because $q \in \mathcal{I}_{\mathcal{L}}$, it follows that $q\ B\ q$ and that $s \in T(\mathcal{G})$ (Definition 5). By Lemma 4 the proof follows.
        - Case $\neg(m(q) = s \wedge m(q') = s' \wedge q \xrightarrow{a}_{\mathcal{L}} q')$. Hence, the transition is not matched on by m and, by Definition 5, remains present in $T(\mathcal{G})$.

     (b) Case $s' \in m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$. Let $q' \in \mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}}$ such that $m(q') = s'$. By $(D2)$ of Definition 4, there exist a $q \in \mathcal{S}_{\mathcal{L}}$ such that $m(q) = s$ and $q \xrightarrow{a}_{\mathcal{L}} q'$. Furthermore, since $s \notin m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$, we have $q \in \mathcal{I}_{\mathcal{L}}$ (by Lemma 1). Because $q \in \mathcal{I}_{\mathcal{L}}$, it follows that $q\ B\ q$ and that $s \in T(\mathcal{G})$ (Definition 5). The proof follows from Lemma 4.

  2. Case $s\ \hat{B}\ t$. We have $s \in m(\mathcal{S}_{\mathcal{L}})$, $s' \in \mathcal{S}_{\mathcal{G}}$, and $s \xrightarrow{a}_{\mathcal{G}} s'$. We distinguish two cases:
     (a) Case $s' \in \mathcal{S}_{\mathcal{G}} \setminus m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$. Again, we distinguish two cases:
        - Case $s \in m(\mathcal{I}_{\mathcal{L}})$. Both s and s' are not removed by the transformation. Since $s\ \hat{B}\ t$, we have $q\ B\ t$ with $m(q) = s$ and $t \in \mathcal{S}_{\mathcal{R}}$. Hence, by Lemma 3, $t \Rightarrow_{\mathcal{R}} s$. By Definition 5, the proof follows: $t \Rightarrow_{T(\mathcal{G})} s \xrightarrow{a}_{T(\mathcal{G})} s'$.
        - Case $s \in m(\mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}})$. Let $q \in \mathcal{S}_{\mathcal{L}} \setminus \mathcal{S}_{\mathcal{R}}$ such that $m(q) = s$. By $(D1)$ of Definition 4, there exist a $q' \in \mathcal{S}_{\mathcal{L}}$ such that $m(q') = s'$ and $q \xrightarrow{a}_{\mathcal{L}} q'$. Because $s\ \hat{B}\ t$, we have $q\ B\ t$. By Lemma 4 the proof follows.

(b) Case $s' \in m(\mathcal{S}_\mathcal{L} \setminus \mathcal{S}_\mathcal{R})$. Let $q' \in \mathcal{S}_\mathcal{L} \setminus \mathcal{S}_\mathcal{R}$ such that $m(q') = s'$. By $(D2)$ of Definition 4, there exist a $q \in \mathcal{S}_\mathcal{L}$ such that $m(q) = s$ and $q \xrightarrow{a}_\mathcal{L} q'$. Because $s \hat{B} t$, we have $q \; B \; t$. The proof follows from Lemma 4.

- If $s \; C \; t$ and $t \xrightarrow{a}_{T(\mathcal{G})} t'$ then either $a = \tau \wedge s \; C \; t'$, or $s \Rightarrow_\mathcal{G} \hat{s} \xrightarrow{a}_\mathcal{G} s' \wedge \hat{s} \; C \; t \wedge s' \; C \; t'$.

  This case has not been mechanically verified. However, this case is symmetrical to the previous case.

□

Lemma 4 states that if there exists a transition $q \xrightarrow{a}_\mathcal{L} q'$ matching $s \xrightarrow{a}_\mathcal{G} s'$, then a state $t \in T(\mathcal{G})$ simulates state $s \in \mathcal{S}_\mathcal{G}$ if state t is matched by a state $p \in \mathcal{S}_\mathcal{R}$ such that $q \; B \; p$.

Lemma 4. *Consider transformation rule r and its matches m and $\hat{m}$ on an LTS $\mathcal{G}$. Let $s, s' \in m(\mathcal{S}_\mathcal{L})$ with $s \xrightarrow{a}_\mathcal{G} s'$, and let $t \in \mathcal{S}_{T(\mathcal{G})}$. Furthermore, let $q, q' \in \mathcal{S}_\mathcal{L}$ such that $m(q) = s$ and $m(q') = s'$, and let $p \in \mathcal{S}_\mathcal{R}$ with $\hat{m}(p) = t$ and $q \; B \; p$. Then $q \xrightarrow{a}_\mathcal{L} q'$ implies:*

- *either $a = \tau$ with $s' \; C \; t$;*
- *or $t \Rightarrow_{T(\mathcal{G})} \hat{t} \xrightarrow{a}_{T(\mathcal{G})} t'$ with $s \; C \; \hat{t}$ and $s' \; C \; t'$.*

Proof. Since $q \; B \; p$, by Definition 2, we have the following two cases:

- Case $a = \tau$ with $q' \; B \; p$. By definition of $\hat{B}$, we have $m(q') \; \hat{B} \; \hat{m}(p)$. Since $m(q') = s'$ and $\hat{m}(p) = t$, we have $s' \; C \; t$.
- Case $p \Rightarrow_\mathcal{R} \hat{p} \xrightarrow{a}_\mathcal{R} p'$ with $q \; B \; \hat{p}$ and $q' \; B \; p'$. By definition of $\hat{B}$, we have $m(q) \; \hat{B} \; \hat{m}(\hat{p})$ and $m(q') \; \hat{B} \; \hat{m}(p')$. Since $m(q) = s$ and $m(q') = s'$, it follows that $s \; C \; \hat{m}(\hat{p})$ and $s' \; C \; \hat{m}(p')$. Furthermore, by Definition 5, we have $t \Rightarrow_{T(\mathcal{G})} \hat{m}(\hat{p}) \xrightarrow{a}_{T(\mathcal{G})} \hat{m}(p')$. Choosing $\hat{t} = \hat{m}(\hat{p})$ and $t' = \hat{m}(p')$, the proof follows.

□

Chapter 4

Transformation of multiple synchronizing processes

Section 4.1 introduces a concurrent system as a network of LTSs. Next, in Section 4.2 we discuss the transformation of a network of LTSs given a system of transformation rules and a vector of matches. Section 4.3 considers the effects of manipulation of synchronising behaviour and how these effects relate to the bisimulation-preservation check. Finally, the proof of the correctness of the bisimulation-preservation check for concurrent systems is given in 4.4. The most noteworthy formalizations in Coq are presented in Section B.2 of Appendix B.

4.1 A concurrent system

Network of LTSs A *network of LTSs* [12], or *LTS network* for short, 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 LTS network is defined as shown in Definition 7. We write $1..n$ for the set of integers ranging from 1 to n. A vector $\bar{v}$ of size n contains n elements indexed from 1 to n. For all $i \in 1..n$, $\bar{v}[i]$ represents the ith element of vector $\bar{v}$.

Definition 7 (Network of LTSs)**.** *A network of LTSs $\mathcal{M}$ of size n is a pair $(\Pi, \mathcal{V})$, where*

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

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

The set of indices of processes participating in a synchronisation law $(\bar{t}, a)$ is formally defined as $Ac(\bar{t}) = \{i \mid i \in 1..n \land \bar{t}[i] \neq \bullet\}$. The set of actions involved in synchronisation laws with multiple processes is given by $\mathcal{A}_s = \{\bar{t}[i] \mid i \in 1..n \land (\bar{t}, a) \in \mathcal{V} \land \bar{t}[i] \neq \bullet \land |Ac(\bar{t})| > 1\}$. The set $\mathcal{A}_s$ contains all synchronising actions.

System LTS of an LTS network. The explicit behaviour of an LTS network $\mathcal{M}$ is described by its system LTS. The *system LTS* combines the processes in Π and the synchronisation laws in $\mathcal{V}$ as shown in Definition 8.

Definition 8 (System LTS of a network of LTSs)**.** *The system LTS* $(\mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, \mathcal{I}_\mathcal{M})$*, describing the behaviour of a network of LTSs* $\mathcal{M}$ *is obtained by combining the processes in* Π *according to the synchronisation laws in* $\mathcal{V}$*:*

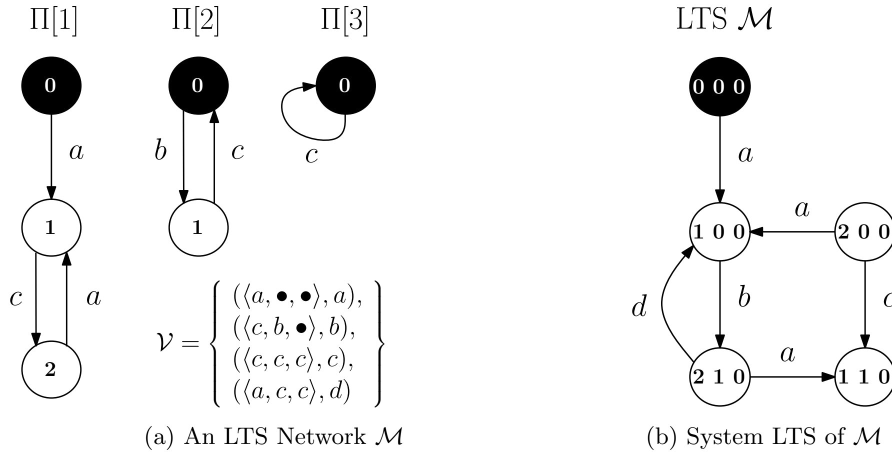
- $\mathcal{S}_\mathcal{M} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$*;*
- $\mathcal{A}_\mathcal{M} = \{a \mid (\bar{t}, a) \in \mathcal{V}\} \cup \{\tau\}$*;*
- $\mathcal{T}_\mathcal{M}$ *is the smallest transition relation satisfying:*

$$\exists(\bar{t}, a) \in \mathcal{V} : \forall i \in 1..n : \begin{pmatrix} (\bar{t}[i] = \bullet \wedge \bar{s}[i] = \bar{s}'[i] \wedge \bar{s}[i] \in \mathcal{S}_i) \\ \vee \; (\bar{t}[i] \neq \bullet \wedge \bar{s}[i] \xrightarrow{\bar{t}[i]}_i \bar{s}'[i]) \end{pmatrix} \implies \bar{s} \xrightarrow{a}_\mathcal{M} \bar{s}';$$

- $\mathcal{I}_\mathcal{M} = \{\langle s_1, \ldots, s_n \rangle \mid \forall i \in 1..n : s_i \in \mathcal{I}_i\}$.

We write $\bar{s} \xrightarrow{\bar{t}\ a}_\mathcal{M} \bar{s}'$ as a shorthand notation for a transition $\bar{s} \xrightarrow{a}_\mathcal{M} \bar{s}'$ that is enabled by a synchronisation law $(\bar{t}, a) \in \mathcal{V}$.

Besides independent actions, an LTS network also allows specification of (multi-party) synchronisations. An example of an LTS network is shown in Figure 4.1a, the corresponding system LTS is presented in Figure 4.1b. The LTS network has three processes that interact with each other. A multi-party synchronisation is specified by $(\langle c, c, c\rangle, c)$, the c-action in the system LTS is the result of the synchronisation of the c-actions of the three processes. Synchronisation law $(\langle c, b, \bullet\rangle, b)$ indicates a two-party synchronisation between process $\Pi[1]$ and process $\Pi[2]$. As specified by $(\langle a, \bullet, \bullet\rangle, a)$, the a-action of process $\Pi[1]$ can be performed independently of the other processes. Furthermore, non-deterministic behaviour can be introduced by specifying multiple rules for the same actions as shown by synchronisation laws $(\langle a, \bullet, \bullet\rangle, a)$ and $(\langle a, c, c\rangle, d)$: in state 2 1 0 the a-action can either be performed individually or the a-action synchronises together with the c-actions from the other processed to produce action d. When process LTS $\Pi[i]$ has an a-transition, but there is no corresponding synchronisation law, then the action cannot actually be performed, we say the action is *cut*.



(a) An LTS Network $\mathcal{M}$ (b) System LTS of $\mathcal{M}$

Figure 4.1: An LTS network and its system LTS

LTS network admissibility. Branching bisimilarity is a congruence for LTS networks if they are *admissible* with respect to τ-transitions [12]. The definition of admisibility for LTS networks is given in Definition 9.

Definition 9 (LTS network Admissibility)**.** *An LTS network* $\mathcal{M} = (\Pi, \mathcal{V})$ *is called* admissible *iff the following holds:*

1. $\forall(\bar{t}, a) \in \mathcal{V} : (\exists i \in 1..n : \bar{t}[i] = \tau \implies Ac(\bar{t}) = \{i\} \wedge a = \tau)$*;*
2. $\forall i \in 1..n : \forall s, s' \in \mathcal{S}_i : (s \xrightarrow{\tau}_i s' \implies \exists(\bar{t}, a) \in \mathcal{V} : \bar{t}[i] = \tau)$*.*

The *first condition* of Definition 9 enforces that τ-actions are always performed independently and that the resulting action is again the τ-action. The *second condition* ensures that there is a synchronisation law for every process in the LTS network that can perform the τ-action, i.e. τ-actions performed by processes in Π are *never cut* by the synchronisation laws. In the remainder of this thesis, we only consider admissible LTS networks. Admissibility of LTS networks is checked by traversing over the sets of synchronisation laws and process LTS transitions.

4.2 Network transformation

A system of transformation rules. A *system of transformation rules*, or a *rule system* for short, allows the transformation of multiple processes at once. Furthermore, a rule system may introduce new synchronisation laws to an LTS network. In Definition 10 the rule system is defined.

Definition 10 (Rule system)**.** *A* rule system $\Sigma = (R, \hat{\mathcal{V}})$ *consists of a set* R *of transformation rules and a set* $\hat{\mathcal{V}}$ *of synchronisation laws introduced by the rule system.*

Intuitively, a rule system describes how a concurrent system is modified to create a new (often more detailed) 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. Let $\mathcal{G} \Rrightarrow_R \mathcal{G}'$ denote the fact that LTS $\mathcal{G}' = T(\mathcal{G})$ can be obtained by applying a rule $r \in R$ on some matches $m : \mathcal{S}_\mathcal{L} \to \mathcal{S}_\mathcal{G}$ and $\hat{m} : \mathcal{S}_\mathcal{R} \to \mathcal{S}_{\mathcal{G}'}$. Moreover, let $\Rrightarrow_R^*$ be the reflexive, transitive closure of $\Rrightarrow_R$. Confluence of rule systems is defined as shown in Definition 11. It is possible to check whether a rule system is confluent [25, 26]. In the remainder of this thesis we will only consider confluent rule systems.

Definition 11 (Confluent rule system)**.** *Consider a rule system* $\Sigma = (R, \hat{\mathcal{V}})$ *and an LTS* $\mathcal{G}$*. Rule system* Σ *is* confluent *iff for all LTSs* $\mathcal{G}_1$*,* $\mathcal{G}_2$ *with* $\mathcal{G} \Rrightarrow_R^* \mathcal{G}_1$ *and* $\mathcal{G} \Rrightarrow_R^* \mathcal{G}_2$*, there exists an LTS* $\mathcal{G}_3$ *such that* $\mathcal{G}_1 \Rrightarrow_R^* \mathcal{G}_3$ *and* $\mathcal{G}_2 \Rrightarrow_R^* \mathcal{G}_3$*.*

An illustration of a confluence for rule systems is presented in Figure 4.2. The rule system consists of two transformation rules r_1 and r_2. Each arrow indicates the application of a given transformation rule $r \in R$ on some matches m and $\hat{m}$. The dotted graph labels and dashed arrows represent a finite number of graphs and rule applications respectively. Any order of rule applications eventually results in graph $\mathcal{G}_z$.

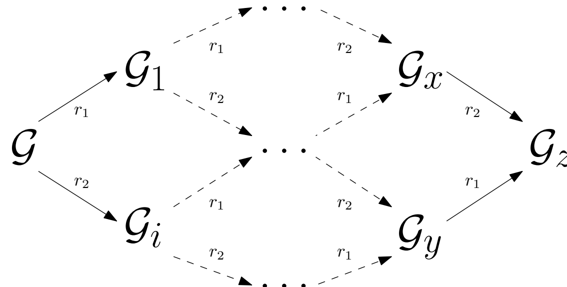


Figure 4.2: Illustration of a confluent rule system

Transformation of an LTS network. An LTS network $\mathcal{M}$ is transformed given a rule system and a number of matches corresponding to rules in the rule system as defined in Definition 12. Each process LTS $\mathcal{G} \in \Pi$ is transformed by applying a transformation rule $r \in R$ according to two corresponding matches $m : \mathcal{S}_{\mathcal{L}} \to \mathcal{S}_{\mathcal{G}}$ and $\hat{m} : \mathcal{S}_{\mathcal{R}} \to \mathcal{S}_{T(\mathcal{G})}$, i.e. each such process LTS $\mathcal{G}$ is transformed obtaining $T(\mathcal{G})$.

An overview of the process transformations is given in Figure 4.3. Definition 12 corresponds to a single transformation-step. Since we assume that rule systems are confluent the order of transformation-steps is irrelevant for the final result. Wijs and Engelen [8, 24, 25] define the transformation of an LTS network with respect to a set of vectors of matches. However, this definition is applied to a scenario that considers only a single vector of matches. Hence, for the sake of consistency, we have defined the transformation of an LTS network as a transformation-step with respect to a single vector of matches rather than a set of vectors of matches.

Definition 12 (LTS network transformation). *Let $\mathcal{M} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \hat{\mathcal{V}})$ be a rule system. Let $\bar{r}$ be a vector of size n such that for all $i \in 1..n$, $\bar{r}[i] \in R$ with corresponding matches $m_i : \mathcal{S}_{\mathcal{L}} \to \mathcal{S}_{\Pi[i]}$ and $\hat{m}_i : \mathcal{S}_{\mathcal{R}} \to \mathcal{S}_{T(\Pi[i])}$. Furthermore, for all $i \in 1..n$ let T_i denote the LTS transformation by application of rule $\bar{r}[i]$ on matches m_i and $\hat{m}_i$ according to Definition 5. The* transformation of LTS network $\mathcal{M}$ *is defined as follows:*

$$T_\Sigma(\mathcal{M}) = (\langle T_1(\Pi[1]), \ldots, T_n(\Pi[n])\rangle, \mathcal{V} \cup \hat{\mathcal{V}})$$

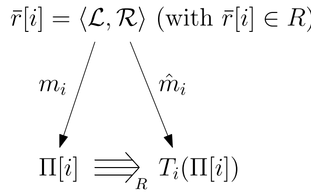


Figure 4.3: Overview of process transformations in Definition 12

Consider an index $i \in 1..n$. We write $\mathcal{L}_i$ and $\mathcal{R}_i$ for the left pattern LTS and the right pattern LTS respectively of transformation rule $\bar{r}[i]$. The set of indices of processes of a network that potentially directly synchronise with behaviour matched by $\mathcal{L}_i$, called the *direct dependency set* of process $\Pi[i]$, is defined by $ddep_{\mathcal{L}}(i) = \bigcup_{(\bar{t},a)\in\mathcal{V}}\{Ac(\bar{t}) \mid \bar{t}[i] \in \mathcal{A}_{\mathcal{L}_i}\}$. An index $j \in 1..n$ is in $ddep_{\mathcal{L}}(i)$ iff there exists a synchronisation law $(\bar{t}, a) \in \mathcal{V}$ such that $i \neq \bullet$ and $j \neq \bullet$. This means both $\Pi[i]$ and $\Pi[j]$ participate in a synchronisation law. For $j \in 1..n$, the set of actions of process $\Pi[j]$ on which $\bar{r}[i]$ depends is given by $A_{ddep_{\mathcal{L}}(i)}(j) = \{\bar{t}[j] \mid (\bar{t}, a) \in \mathcal{V} \wedge \bar{t}[i] \in \mathcal{A}_{\mathcal{L}_i} \wedge \bar{t}[j] \neq \bullet\}$. An action $b \in \mathcal{A}_{\Pi[j]}$ is in $A_{ddep_{\mathcal{L}}(i)}(j)$ iff there exists an action $c \in \mathcal{A}_{\mathcal{L}_i}$ with which the b-action synchronises. The direct dependency set extended with indirectly dependent process indices, called the *dependency set* of process $\Pi[i]$, is defined by the transitive closure of $ddep_{\mathcal{L}}$, i.e. $dep_{\mathcal{L}}(i) = ddep^+_{\mathcal{L}}$. Sets $ddep_{\mathcal{R}}(i)$, $A_{ddep_{\mathcal{R}}(i)}(j)$, and $dep_{\mathcal{R}}(i)$ are defined similarly using the synchronisation laws $\mathcal{V} \cup \hat{\mathcal{V}}$. Finally, the combined dependency set of $\Pi[i]$ and $T_i(\Pi[i])$ is defined as $dep(i) = dep_{\mathcal{L}}(i) \cup dep_{\mathcal{R}}(i)$.

Consider an LTS network $\mathcal{M}$. States in the system LTS of $\mathcal{M}$ are vectors $\bar{s}$ of size n. Any element in such a state can be matched on by some transformation rule. The set defined as $M(\bar{s}) = \{i \mid \bar{s}[i] \in m_i(\mathcal{S}_{\mathcal{L}_i})\}$, is the set of indices of elements in $\bar{s}$ matched on by the corresponding transformation rule in $\bar{r}$.

4.3 Transformation of synchronising behaviour

Synchronisation properties. To ease reasoning about transformation of synchronising behaviour we introduce a few restrictions. These restrictions, called *synchronisation uniformity*, are defined in Definition 13.

Definition 13 (Synchronisation uniformity)**.** *A rule system* $\Sigma = (R, \hat{\mathcal{V}})$ *is called* synchronisation uniform *with respect to LTS network* $\mathcal{M}$ *iff the following holds:*

1. $\forall a \in \mathcal{A}_s, i \in 1..n : (\exists r \in R : a \in \mathcal{A}_{\mathcal{L}_i}) \implies$
   $\forall s \xrightarrow{a}_i s' : \exists q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = s \wedge m_i(q') = s' \wedge q \xrightarrow{a}_{\mathcal{L}_i} q'$;
2. $\forall a \in \mathcal{A}_s, i \in 1..n : (\exists r \in R : a \in \mathcal{A}_{\mathcal{R}_i}) \implies$
   $\forall s \xrightarrow{a}_{T_i(\Pi[i])} s' : \exists q, q' \in \mathcal{S}_{\mathcal{R}_i} : \hat{m}_i(q) = s \wedge \hat{m}_i(q') = s' \wedge q \xrightarrow{a}_{\mathcal{R}_i} q'$;
3. $\forall i \in 1..n, j \in dep_{\mathcal{L}}(i) : A_{ddep_{\mathcal{L}}(i)}(j) \subseteq \mathcal{A}_{\mathcal{L}_j}$;
4. $\forall i \in 1..n, j \in dep_{\mathcal{R}}(i) : A_{ddep_{\mathcal{R}}(i)}(j) \subseteq \mathcal{A}_{\mathcal{R}_j}$;
5. $\forall (\bar{t}, a) \in \hat{\mathcal{V}}, i \in 1..n : \bar{t}[i] \notin \mathcal{A}_i$.

The *first* and *second condition* are called the *universal applicability* conditions. This condition expresses that a transformation rule matching a synchronising transition must match all synchronising transitions with that label. Without constructing the state space of an LTS network it is difficult to determine which transitions in the different processes will be able to synchronise. If these conditions would not hold, the transformed and unchanged transitions may follow a different communication protocol. It is hard to distinguish these transformed and unchanged transitions during verification of the rule system. As a consequence, it is difficult to determine whether the transformed network is bisimilar to the original network by reasoning about the rule system alone.

The *third* and *fourth condition* state that a rule system must be *complete* with respect to dependent actions. All actions that can synchronise with behaviour of $\mathcal{L}_i$ (or $\mathcal{R}_i$) must also be transformed by some transformation rule in R. Should this condition not hold, it becomes difficult to reason about synchronising behaviour between LTS patterns. If some of the synchronising parties are changed while others are not, it is difficult to say how this affects the unchanged synchronising parties.

The *fifth condition* enforces that new synchronisation laws only involve actions that are not present in the corresponding source LTS. This condition prevents new synchronisation laws from changing the semantics of a model in a way that is inconsistent with the transformation rules.

The second and fourth condition are new with respect to [8, 24, 25]. These conditions are the symmetric equivalents of the first and third conditions respectively. Symmetry of branching bisimulation's transfer conditions requires that these symmetric equivalents are present. By checking the appropriate sets of transitions and actions synchronisation uniformity of rule systems can be determined. In the remainder of this work we only consider synchronously uniform rule systems.

Synchronisation uniformity makes it easier to reason about synchronising behaviour. As expressed in Lemma 5, whenever a transition in the process of a network is matched, all participating transitions of the other processes must be matched as well.

Lemma 5. *Consider an LTS network* $\mathcal{M} = (\Pi, \mathcal{V})$ *of size n. Let* $\bar{r}$ *be a vector of transformation rules given by a network transformation (Definition 12) applied to* $\mathcal{M}$*, then*

$$\begin{aligned}&\forall \bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{M}} : \bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}} \bar{s}' \wedge \\ &\quad (\exists i \in Ac(\bar{t}), q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} q') \implies \\ &\quad\quad (\forall i \in Ac(\bar{t}) : \exists q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} q')\end{aligned}$$

Proof. Let $\bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}} \bar{s}'$. Furthermore, let $j \in Ac(\bar{t})$, $q, q' \in \mathcal{S}_{\mathcal{L}_j}$ such that there is a transition $q \xrightarrow{\bar{t}[j]}_{\mathcal{L}_j} q'$ matching $\bar{s}[j] \xrightarrow{\bar{t}[j]}_{\Pi[j]} \bar{s}'[j]$. If $|\bar{t}| = 1$, then $Ac(\bar{t}) = \{j\}$ and the proof follows from the premises. If $|\bar{t}| > 1$, then $\bar{t}[i] \in \mathcal{A}_s$. Let $i \in Ac(\bar{t})$, we have $\bar{t}[i] \in A_{ddep_{\mathcal{L}}(j)}(i)$. Hence, by condition 3 of Definition 13, we have $\bar{t}[i] \in \mathcal{A}_{\mathcal{L}_i}$. Moreover, from Definition 8 it

follows that $\bar{s}[i] \xrightarrow{a}_{\Pi[i]} \bar{s}'[i]$. By condition 1 of Definition 13 there exists $p, p' \in \mathcal{S}_{\mathcal{L}_i}$ such that there is a transition $p \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} p'$ with $m_i(p) = \bar{s}[i]$ and $m_i(p') = \bar{s}'[i]$. In both cases we have $\forall i \in Ac(\bar{t}) : \exists q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} q'$. □

Consider a confluent rule system $\Sigma = (R, \mathcal{V})$. Since Σ is confluent, it is possible to construct a set of transformation rules R^* from R that $\Sigma^* = (R^*, \hat{\mathcal{V}})$ produces the same output as the exhaustive application of Σ on an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$. This set of transformation rules R^* is created by combining transformation rules in R. Since the left and right LTS patterns of all $r \in R$ are bisimilar, by congruence, all left and right LTS patterns of all $r^* \in R^*$ are bisimilar. Therefore, synchronisation uniformity may be enforced over a sequence of transformation steps rather than a single transformation step as defined by Definition 12, i.e. synchronisation uniformity may be weakened such that it applies to a sequence of transformation steps.

In the remainder of this thesis we assume that a rule system is synchronisation uniform with respect to the model being transformed. When transforming synchronising behaviour, it is natural to transform all involved parties. Thus, it is reasonable to make this assumption.

Networks of transformation rules. When a rule system affects synchronising actions multiple transformation rules must be involved to update the affected processes. By considering appropriate LTS-pattern combinations it is possible to determine whether the combined behaviour of the processes remains bisimilar.

Let $\bar{r}$ be the vector of transformation rules given by a network transformation. Recall that $\mathcal{L}_i$ and $\mathcal{R}_i$ represent the left and right LTS patterns of rule $\bar{r}[i]$ respectively. To analyse combinations of LTS-patterns we first convert the transformation rules in two LTS networks $\mathcal{L}_{\bar{r}} = (\langle \mathcal{L}_1, \dots, \mathcal{L}_n \rangle, \mathcal{V})$ and $\mathcal{R}_{\bar{r}} = (\langle \mathcal{R}_1, \dots, \mathcal{R}_n \rangle, \mathcal{V} \cup \hat{\mathcal{V}})$, called *pattern networks*. Next, the pattern networks are filtered as defined by Definition 14 to produce pattern networks that consider only a given subset of rules. Given a set of indices I, the filtered left and right pattern networks are denoted $\mathcal{L}_I$ and $\mathcal{R}_I$. The filtered pattern networks are equivalent to the notion of rule patterns described in [8]. With these filtered pattern networks the synchronising behaviour before and after transformation can be compared.

Definition 14 (Filtered LTS network)**.** *Given an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$ of size n and a set of indices $I \subseteq 1..n$, the filtered LTS network is defined by $\mathcal{M}_I = (\Phi, \mathcal{V})$, with*

$$\forall i \in 1..n : \Phi[i] = \begin{cases} \Pi[i] & \text{if } i \in I \\ \Delta(\Pi[i]) & \text{otherwise} \end{cases}$$

where $\Delta(\Pi[i]) = (\{*\}, \mathcal{A}_{\Pi[i]}, \emptyset, \{*\})$ is a dummy LTS with a dummy state $*$.

To produce an LTS network the actions of the synchronisation vectors must be elements of the respective LTS. Therefore, the dummy LTS $\Delta(i)$ contains the same action set as the LTS $\Pi[i]$ that the dummy replaces. However, as dummy LTSs represent no behaviour (i.e. the behaviour is filtered), they do not contain any transitions.

Like a single transformation rule, the filtered left and right pattern networks have glue-states as well. In fact, if a filtered left pattern network $\mathcal{L}_I$ matches on $\mathcal{M}$, by Definition 12, matched states in processes indexed by I are replaced by the matched states of the corresponding filtered right pattern network $\mathcal{R}_I$ relative to the glue-states. Lemma 6 states that the glue-states of $\mathcal{L}_I$ and $\mathcal{R}_I$ are those (vector-)states that are present in both filtered pattern networks, i.e. $\mathcal{I}_{\mathcal{L}_I} = \mathcal{I}_{\mathcal{R}_I} \wedge \mathcal{I}_{\mathcal{L}_I} = \mathcal{S}_{\mathcal{L}_I} \cap \mathcal{S}_{\mathcal{R}_I}$.

Lemma 6. *Let $\bar{r}$ be a vector of size n of transformation rules given by a network transformation (Definition 12). Furthermore, Let $\mathcal{L}_I$ and $\mathcal{R}_I$ be the corresponding filtered left and right pattern networks with $I \subseteq 1..n$. The following holds:*

$$\mathcal{I}_{\mathcal{L}_I} = \mathcal{I}_{\mathcal{R}_I} \wedge \mathcal{I}_{\mathcal{L}_I} = \mathcal{S}_{\mathcal{L}_I} \cap \mathcal{S}_{\mathcal{R}_I}$$

Proof. We have to show $\forall \bar{s} \in \mathcal{I}_{\mathcal{L}_I} : \bar{s} \in \mathcal{I}_{\mathcal{R}_I} \wedge \bar{s} \in \mathcal{S}_{\mathcal{L}_I} \cap \mathcal{S}_{\mathcal{R}_I}$, $\forall \bar{s} \in \mathcal{I}_{\mathcal{R}_I} : \bar{s} \in \mathcal{I}_{\mathcal{L}_I}$, and $\forall \bar{s} \in \mathcal{S}_{\mathcal{L}_I} \cap \mathcal{S}_{\mathcal{R}_I} : \bar{s} \in \mathcal{I}_{\mathcal{L}_I}$. The proof for all three cases is analogue. For the sake of brevity, we only present the proof for the first case.

Let $\bar{s} \in \mathcal{I}_{\mathcal{L}_I}$. By Definition 14, we have $\forall i \in I : \bar{s}[i] \in \mathcal{I}_{\mathcal{L}_i}$ and $\forall i \in 1..n \setminus I : \bar{s}[i] \in \mathcal{I}_{\Delta(i)}$. We distinguish two cases. Let $i \in 1..n$:

- Case: $i \in I$. We have $\mathcal{L}_I[i] = \mathcal{L}_i$, $\mathcal{R}_I[i] = \mathcal{R}_i$, and $\bar{r}[i] = (\mathcal{L}_i, \mathcal{R}_i)$. By Definition 3, it follows that $\bar{s}[i] \in \mathcal{I}_{\mathcal{R}_i}$ and $\bar{s}[i] \in \mathcal{S}_{\mathcal{L}_i} \cap \mathcal{S}_{\mathcal{R}_i}$. Hence, $\bar{s}[i] \in \mathcal{R}_I[i]$ and $\bar{s}[i] \in \mathcal{S}_{\mathcal{L}_I[i]} \cap \mathcal{S}_{\mathcal{R}_I[i]}$.
- Case: $i \notin I$. By Definition 14, we have $\mathcal{L}_I[i] = \Delta(\mathcal{L}_i)$ and $\mathcal{R}_I[i] = \Delta(\mathcal{R}_i)$, and $\mathcal{I}_{\Delta(\mathcal{L}_i)} = \mathcal{S}_{\Delta(\mathcal{L}_i)} = \mathcal{S}_{\Delta(\mathcal{R}_i)} = \mathcal{I}_{\Delta(\mathcal{R}_i)} = \{*\}$. Since $\bar{s}[i] = *$, it follows that $\bar{s}[i] \in \mathcal{R}_I[i]$ and $\bar{s}[i] \in \mathcal{S}_{\mathcal{L}_I[i]} \cap \mathcal{S}_{\mathcal{R}_I[i]}$.

In both cases $\bar{s}[i] \in \mathcal{R}_I[i]$ and $\bar{s}[i] \in \mathcal{S}_{\mathcal{L}_I[i]} \cap \mathcal{S}_{\mathcal{R}_I[i]}$. Therefore, $\bar{s} \in \mathcal{I}_{\mathcal{R}_I}$ and $\bar{s} \in \mathcal{S}_{\mathcal{L}_I} \cap \mathcal{S}_{\mathcal{R}_I}$. □

Similar to matches for a single rule, we can define how state vectors of pattern networks are mapped to state vectors of LTS networks. By referring to matches of the individual vector elements a state vector can be matched on to another state vector. Since pattern networks may contain the dummy LTS $\Delta(i)$ a vector state in a pattern network might match on multiple vector states of an LTS network. This mapping is defined as presented in Definition 15.

Definition 15 (Mapping of state vectors)**.** *Consider an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$ of size n and a filtered LTS network $\mathcal{N}_I = (\Phi, \mathcal{V})$ with $I \subseteq 1..n$. Let $\bar{r}$ be a vector of transformation rules with for each $\bar{r}[i]$ a corresponding match $m_i : \mathcal{S}_{\Phi[i]} \to \mathcal{S}_{\Pi[i]}$. A state $\bar{q} \in \mathcal{S}_{\mathcal{N}_I}$ is mapped to a state $\bar{s} \in \mathcal{S}_{\mathcal{M}}$, denoted by $\bar{q} \vdash_I \bar{s}$, iff*

$$\forall i \in I : m_i(\bar{q}[i]) = \bar{s}[i]$$

The presented mapping amounts to a simulation relation between state vectors. For state vectors $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ and $\bar{p} \in \mathcal{S}_{\mathcal{N}_I}$, if $\bar{p} \vdash_I \bar{s}$, then the group of states $\bar{s}[i]$ indexed by $i \in I$ can simulate the behaviour of the state vector $\bar{p}$. This concept is formalized in Lemma 7 and Lemma 8.

Lemma 7. *Let $\mathcal{M} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\mathcal{N}_I = (\Phi, \mathcal{V})$ be a filtered pattern network, with $I \subseteq 1..n$. Let $\bar{r}$ be a vector of transformation rules with for each $\bar{r}[i]$ a corresponding match $m_i : \mathcal{S}_{\Phi[i]} \to \mathcal{S}_{\Pi[i]}$, then*

$$\begin{aligned}\forall \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{N}_I} : \bar{p} \xrightarrow{a}_{\mathcal{N}_I} \bar{p}' \implies (\forall \bar{s} \in \mathcal{S}_{\mathcal{M}} : \bar{p} \vdash_I \bar{s} \implies \\ \exists \bar{s}' \in \mathcal{S}_{\mathcal{M}} : \bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}' \wedge \bar{p}' \vdash_I \bar{s}' \wedge \forall i \in 1..n \setminus I : \bar{s}[i] = \bar{s}'[i])\end{aligned}$$

Proof. Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{N}_I}$ with $\bar{p} \xrightarrow{a}_{\mathcal{N}_I} \bar{p}'$, and let $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{p} \vdash_I \bar{s}$. Let $(\bar{t}, a) \in \mathcal{V}$ be the synchronisation law such that $\bar{p} \xrightarrow{\bar{t}\ a}_{\mathcal{N}_I} \bar{p}'$. We show how to construct a state $\bar{s}' \in \mathcal{S}_{\mathcal{M}}$ according to Definitions 14 and 15 such that $\bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}} \bar{s}'$, $\bar{p}' \vdash_I \bar{s}'$, and $\forall i \in 1..n \setminus I : \bar{s}[i] = \bar{s}'[i]$, i.e. we show:

$$\begin{aligned}\forall i \in 1..n : \exists q' \in \mathcal{S}_i : (i \in Ac(\bar{t}) \implies \bar{s}[i] \xrightarrow{\bar{t}[i]}_i q') \wedge (i \notin Ac(\bar{t}) \implies \bar{s}[i] = q') \\ \wedge (i \in I \implies m_i(\bar{p}'[i]) = q') \wedge (i \notin I \implies \bar{s}[i] = q')\end{aligned}$$

Since $\bar{p} \xrightarrow{\bar{t}\ a}_{\mathcal{N}_I} \bar{p}'$, all active sub-states are part of the filtered pattern network. It follows that $Ac(\bar{t}) \subseteq I$. Let $i \in 1..n$, we distinguishing three cases:

- Case: $i \in I \cap Ac(\bar{t})$. We have to show $\exists q' \in \mathcal{S}_i : m_i(\bar{p}') = q' \wedge \bar{s}[i] \xrightarrow{\bar{t}[i]}_i q')$.

  By condition 3 of Definition 4, it follows that $m_i(\bar{p}[i]) \xrightarrow{\bar{t}[i]}_i m_i(\bar{p}'[i])$. Since a match is an injective function, we have $m_i(\bar{p}[i]) = \bar{s}[i]$. Hence, we get $\bar{s}[i] \xrightarrow{\bar{t}[i]}_{\mathcal{M}} m_i(\bar{p}'[i])$. It follows that $m_i(\bar{p}'[i])$ is the appropriate choice for q'.

- Case: $i \in I \setminus Ac(\bar{t})$. We have to show $\exists q' \in \mathcal{S}_i : m_i(\bar{p}') = q' \wedge q' = \bar{s}[i]$.

  Since $i \notin Ac(\bar{t})$ we have $\bar{p}[i] = \bar{p}'[i]$. It follows that $m_i(\bar{p}') = m_i(\bar{p}) = \bar{s}[i]$. Hence, a suitable choice is $q' = \bar{s}[i]$.

- Case: $i \notin I$. We have to show there is a $q' \in \mathcal{S}_i$ such that $q' = \bar{s}[i]$.

  Since $Ac(\bar{t}) \subseteq I$, it follows that $i \notin Ac(\bar{t})$. Hence, by Definition 8, we can take $q' = \bar{s}[i]$.

It follows that there exists an $\bar{s}' \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{s} \xrightarrow{a} \bar{s}'$, $\bar{p}' \vdash_I \bar{s}'$, and $\forall i \in 1..n \setminus I : \bar{s}[i] = \bar{s}'[i]$. □

Lemma 8. *Let $\mathcal{M} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\mathcal{N}_I = (\Phi, \mathcal{V})$ be a filtered pattern network, with $I \subseteq 1..n$. Let $\bar{r}$ be a vector of transformation rules with for each $\bar{r}[i]$ a corresponding match $m_i : \mathcal{S}_{\Phi[i]} \to \mathcal{S}_{\Pi[i]}$, then*

$$\forall \bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{N}_I} : (\bar{p} \Rightarrow_{\mathcal{N}_I} \bar{p}') \implies (\forall \bar{s} \in \mathcal{S}_{\mathcal{M}} : \bar{p} \vdash_I \bar{s} \implies \exists \bar{s}' \in \mathcal{S}_{\mathcal{M}} : (\bar{s} \Rightarrow_{\mathcal{M}} \bar{s}') \wedge \bar{p}' \vdash_I \bar{s}' \wedge \forall i \in 1..n \setminus I : \bar{s}[i] = \bar{s}'[i])$$

Proof. Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{N}_I}$ with $\bar{p} \Rightarrow_{\mathcal{N}_I} \bar{p}'$, and let $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{p} \vdash_I \bar{s}$. We show how to construct a state $\bar{s}' \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{s} \Rightarrow_{\mathcal{M}} \bar{s}'$, $\bar{p}' \vdash_I \bar{s}'$, and $\forall i \in 1..n \setminus I : \bar{s}[i] = \bar{s}'[i]$ by structural induction on $\Rightarrow_{\mathcal{N}_I}$:

- Base case: $\bar{p} \Rightarrow_{\mathcal{N}_I} \bar{p}$. We have to show $\bar{s} \Rightarrow_{\mathcal{M}} \bar{s}$, this follows from reflexivity of $\Rightarrow$.

- Base case: $\bar{p} \xrightarrow{\tau}_{\mathcal{N}_I} \bar{p}'$. We have to show $\exists \bar{s}' \in \mathcal{S}_{\mathcal{M}} : \bar{s} \xrightarrow{\tau}_{\mathcal{M}} \bar{s}'$, this follows directly from Lemma 7.

- Step case: $\bar{p} \Rightarrow_{\mathcal{N}_I} \hat{\bar{p}}$ and $\hat{\bar{p}} \Rightarrow_{\mathcal{N}_I} \bar{p}'$. We have to show $\bar{s} \Rightarrow_{\mathcal{M}} \bar{s}'$.

  By the Induction Hypothesis we have $\bar{s} \Rightarrow_{\mathcal{M}} \hat{\bar{s}}$ and $\hat{\bar{s}} \Rightarrow_{\mathcal{M}} \bar{s}'$. The proof follows from the transitivity of $\Rightarrow$.

□

Consider a set of indices $I \subseteq 1..n$ and a state vector $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ of some LTS network $\mathcal{M}$. Lemma 9 states the following: when states $\bar{s}[i]$ indexed by $i \in I$ are matched there is a state vector $\bar{q} \in \mathcal{S}_{\mathcal{L}_I}$ such that $\bar{q} \vdash_I \bar{s}$.

Lemma 9. *Consider an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$ and a filtered left-pattern network $\mathcal{L}_I = (\mathcal{L}, \mathcal{V})$ with $I \subseteq 1..n$. Let $\bar{r}$ be a vector of transformation rules with for each $\bar{r}[i]$ a corresponding match $m_i : \mathcal{S}_{\mathcal{L}_i} \to \mathcal{S}_{\Pi[i]}$.*

$$\forall s \in \mathcal{S}_{\mathcal{M}} : (\forall i \in I : i \in M(\bar{s})) \implies \exists \bar{q} \in \mathcal{S}_{\mathcal{L}_I} : \ \bar{q} \vdash_I \bar{s}$$

Proof. Let $s \in \mathcal{S}_{\mathcal{M}}$ with $(\forall i \in I : i \in M(\bar{s}))$. We have to show that there is a $\bar{q} \in \mathcal{S}_{\mathcal{L}_I}$ such that $\bar{q} \vdash_I \bar{s}$. By Definition 15, it is sufficient to show that for all $i \in I$ there exists a $q \in \mathcal{S}_{\mathcal{L}_i}$ such that $m_i(\bar{q}[i]) = \bar{s}[i]$. Since $(\forall i \in I : i \in M(\bar{s}))$, by definition of $M(\bar{s})$, for all $i \in I$ there exists such a state $q \in \mathcal{S}_{\mathcal{L}_i}$ with $m_i(\bar{q}[i]) = \bar{s}[i]$. □

Previously, in Lemma 5, it was shown that all transitions participating in a synchronization law must be matched if there is a match on one of the participating transitions. Lemma 10 formulates a similar statement on the level of transitions in a system LTS: whenever a transition participating in synchronisation is matched, then there is a (filtered) pattern network matching the corresponding transition in the system LTS.

Lemma 10. *Consider an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$ of size n. Let $\bar{r}$ be a vector of transformation rules given by a network transformation (Definition 12) applied to $\mathcal{M}$, then*

$$\forall \bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{M}} : \bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}} \bar{s}' \wedge$$
$$(\exists i \in Ac(\bar{t}), q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} q') \implies$$
$$(\forall \bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}} : \bar{s}_m \vdash_{M(\bar{s})} \bar{s} \implies \exists \bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}} : \bar{s}'_m \vdash_{M(\bar{s})} \bar{s}' \wedge \bar{s}_m \xrightarrow{\bar{t}\ a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m)$$

Proof. Let $\bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{M}}$ such that $\bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}}$. Furthermore, let $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ such that $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$. We have to show $\exists \bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}} : \bar{s}'_m \vdash_{M(\bar{s})} \bar{s}' \wedge \bar{s}_m \xrightarrow{\bar{t}\ a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$. We show how to construct $\bar{s}'_m$ according to Definitions 14 and 15, i.e. we show $\forall i \in M(\bar{s}) : \exists p' \in \mathcal{S}_{\mathcal{L}_i} : m_i(p') = \bar{s}'[i] \wedge (i \in Ac(\bar{t}) \implies \bar{s}_m[i] \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} p')$. Note that for $i \in 1..n \setminus M(\bar{s})$ we have $\bar{s}'_m[i] = *$. Hence, there are no proof obligations and the case can be admitted.

Let $i \in M(\bar{s})$. We distinguish two cases:

- Case: $i \in Ac(\bar{t})$. We have to show that $\exists p' \in \mathcal{S}_{\mathcal{L}_i} : m_i(p') = \bar{s}'[i] \wedge \bar{s}_m[i] \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} p'$.

  By Lemma 5 there exist $p, p' \in \mathcal{S}_{\mathcal{L}_i} : m_i(p) = \bar{s}[i]$, $m_i(p') = \bar{s}'[i]$, and $p \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} p'$. From the injectivity property of Definition 4 it follows that $p = \bar{s}_m[i]$. We get $\bar{s}_m[i] \xrightarrow{\bar{t}[i]}_{\mathcal{L}_i} p'$. Thus, p' is a fitting choice.

- Case: $i \notin Ac(\bar{t})$. We have to show that $\exists p' \in \mathcal{S}_{\mathcal{L}_i} : m_i(p') = \bar{s}'[i]$.

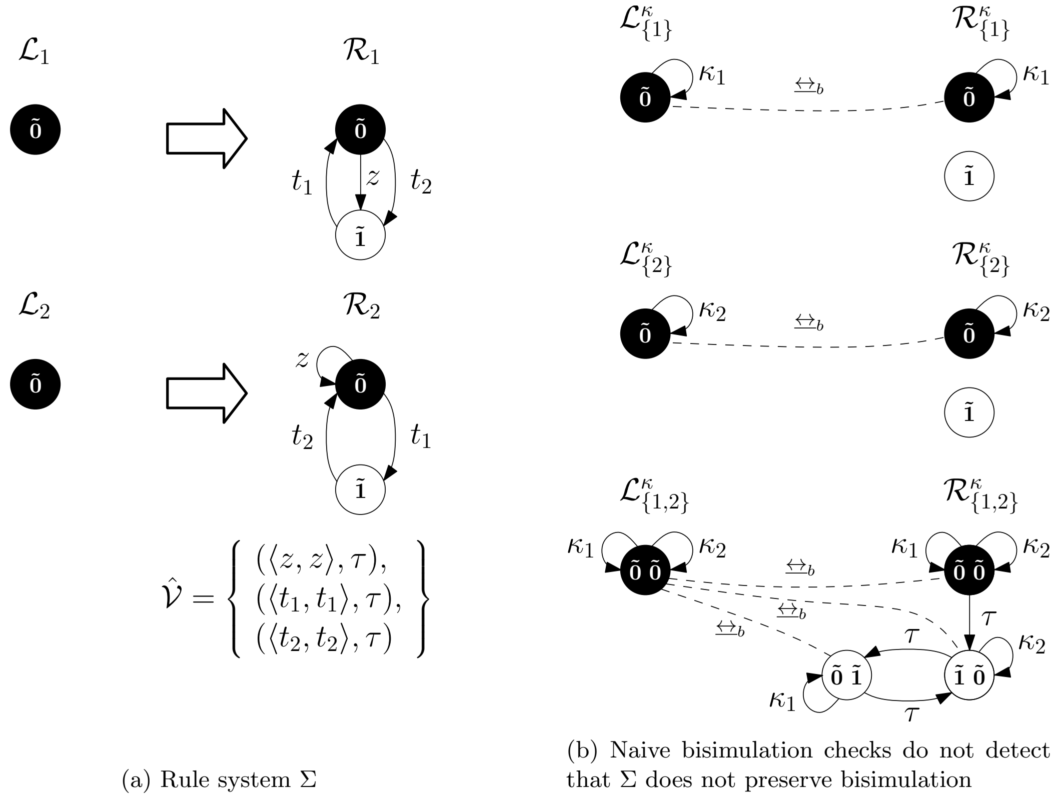
  Following Definition 8, we have $\bar{s}[i] = \bar{s}'[i]$. It follows that $m_i(\bar{s}_m[i]) = \bar{s}[i] = \bar{s}'[i]$. Therefore, we take $p' = \bar{s}_m[i]$.

Hence, for all $i \in M(\bar{s})$ there is a $p'_i \in \mathcal{S}_{\mathcal{L}_i}$ such that $m_i(p'_i) = \bar{s}'_m[i]$. Additionally, we have $\bar{s}_m \xrightarrow{\bar{t}\ a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$. □

Handling incoming and outgoing synchronisation for pattern networks. When (filtered) pattern networks are formed it is also important to consider incoming and outgoing synchronising transitions that may not be present in the pattern network. Before we formally introduce the κ-extension $\mathcal{M}^\kappa = (\Pi^\kappa, \mathcal{V}^\kappa)$ of an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$ we first explain why incoming and outgoing synchronising transitions must be considered with an example.

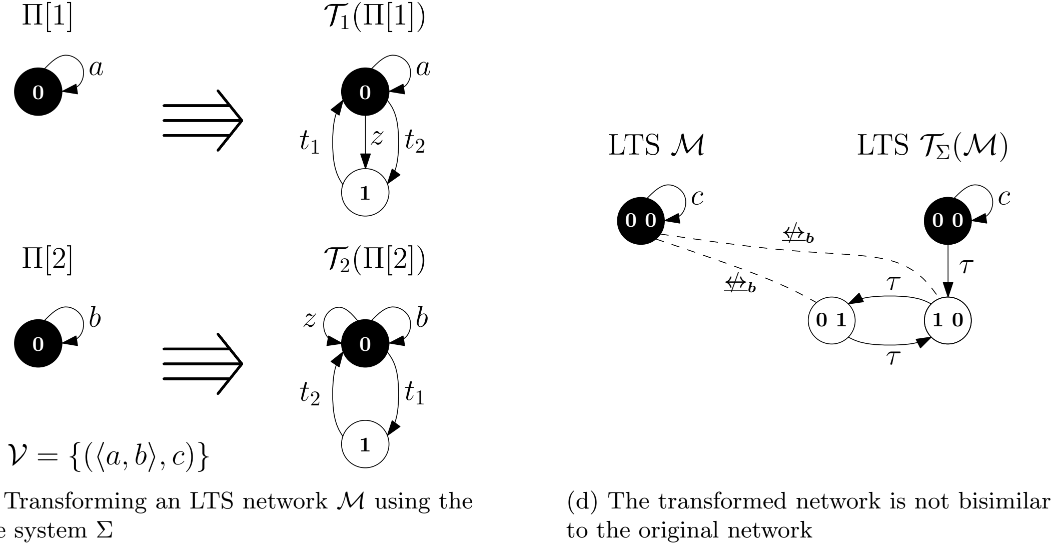
As illustrated in Figure 4.4, it is not sufficient to only consider the κ-self loops in extended transformation rules (Definition 6). Figure 4.4a presents a rule system that does not preserve bisimulation. Again, the states are numbered such that matches can be identified by the state label, i.e. a state $\tilde{i}$ is matched onto state i of the corresponding process LTS of Figure 4.4c. Figure 4.4b shows that individual κ-self loops are not sufficient to detect that the rule system is not bisimulation preserving. In Figure 4.4c LTS network $\mathcal{M}$ is transformed using the rule system of Figure 4.4a. The system LTSs before (LTS $\mathcal{M}$) and after (LTS $T_\Sigma(\mathcal{M})$) applying the rule system are presented in Figure 4.4d. Clearly, these system LTSs are not bisimilar. However, the failure to preserve bisimulation can be detected if an additional transition representing the synchronisation of κ_1 and κ_2 is added in the pattern networks $\mathcal{L}^\kappa_{\{1,2\}}$ and $\mathcal{R}^\kappa_{\{1,2\}}$. States $(\tilde{0}\ \tilde{1})$ and $(\tilde{1}\ \tilde{0})$ would not be able to perform this κ-synchronisation, and thus $\mathcal{L}^\kappa_{\{1,2\}} \not\leftrightarrow_b \mathcal{R}^\kappa_{\{1,2\}}$.

The κ-extensions for LTS networks is formally defined in Definition 16. The filtered κ-extended left and right pattern networks are denoted $\mathcal{L}^\kappa_I$ and $\mathcal{R}^\kappa_I$ respectively, they are created by first applying κ-extension on the pattern networks and then filtering the κ-extended pattern networks.



(a) Rule system Σ

(b) Naive bisimulation checks do not detect that Σ does not preserve bisimulation



(c) Transforming an LTS network $\mathcal{M}$ using the rule system Σ

(d) The transformed network is not bisimilar to the original network

Figure 4.4: The synchronisation of κ-self loops must be considered to properly determine bisimulation preservation for rule systems

Definition 16 (Extension of an LTS network with κ-synchronisation)**.** *Consider an LTS network* $\mathcal{M} = (\Pi, \mathcal{V})$*. The* κ-extension *of* $\mathcal{M}$ *is defined as* $\mathcal{M}^\kappa = (\Pi^\kappa, \mathcal{V}^\kappa)$*, with*

- $\forall i \in 1..n : \Pi^\kappa[i] = (\Pi[i])^\kappa$
- $\mathcal{V}^\kappa = \mathcal{V} \cup \{(\langle \kappa_1, \ldots, \kappa_n\rangle, \kappa_{\bar{s}}) \mid \bar{s} \in \mathcal{I}_\mathcal{M} \wedge (\forall i \in 1..n : \kappa_i = \kappa_{\bar{s}[i]} \vee \kappa_i = \bullet)\} \setminus \{\langle \bullet, \ldots, \bullet\rangle\}$*, where* $\kappa_{\bar{s}}$ *is a unique result label such that* $\forall (\bar{t}, a) \in \mathcal{V}^\kappa : a = \kappa_{\bar{s}} \implies \bar{t} = \langle \kappa_1, \ldots, \kappa_n\rangle$.

The κ-synchronisation laws ensure that each vector of glue-states is at least related to itself. Furthermore, the κ-synchronisation laws prevent that a vector state $\bar{s}$ which consists of a number of glue-states is related to a vector state $\bar{q}$ consisting of the same glue-states plus some additional glue states if $\bar{s}$ cannot reach such a state with the same glue-sates as $\bar{q}$, i.e. the situation presented in Figure 4.4 is prevented. For each $\bar{s} \in \mathcal{I}_{\mathcal{M}}$ there is a number of κ-synchronisation laws that represent incoming and outgoing synchronising transitions for combinations of κ actions that can be performed by vector glue-state $\bar{s}$. Each κ-synchronisation law v has a unique result label, i.e. κ-synchronisation laws can be identified distinguished based solely on their result action. The synchronisation of two κ-actions κ_i and κ_j will be denoted by the κ_{ij}-action. The introduction of κ-synchronisation laws is a novel contribution to work of Wijs and Engelen [8, 24, 25].

The κ-synchronisation actions are unique for each vector of glue-states. Hence, as stated in Lemma 11, each vector of glue-states must be related to itself.

Lemma 11. *Consider a vector of transformation rules $\bar{r}$ of size n given by a network transformation (Definition 12). Let $I \subseteq 1..n$ be a set of indices, and let $\mathcal{L}_I$ and $\mathcal{R}_I$ be filtered left and right pattern networks produced from $\bar{r}$. Let B_I be a branching bisimulation relation such that $\mathcal{L}_I^\kappa \leftrightarrow_b \mathcal{R}_I^\kappa$, then*

$$\forall \bar{s} \in \mathcal{I}_{\mathcal{L}_I} : \bar{s}\ B_I\ \bar{s}$$

Proof. Let $\bar{s} \in \mathcal{I}_{\mathcal{L}_I}$, by Definition 16, we have $\bar{s} \in \mathcal{I}_{\mathcal{L}_I^\kappa}$. Since $\mathcal{L}_I^\kappa \leftrightarrow_b \mathcal{R}_I^\kappa$, there exists a $\bar{p} \in \mathcal{I}_{\mathcal{R}_I^\kappa}$ such that $\bar{s}\ B_I\ \bar{p}$. From Definition 16 it follows that $\bar{s} \xrightarrow{\kappa_{\bar{s}}}_{\mathcal{L}_I^\kappa} \bar{s}$. Moreover, since $\bar{s}\ B_I\ \bar{p}$, we also have $\bar{p} \Rightarrow_{\mathcal{R}_I^\kappa} \hat{\bar{p}} \xrightarrow{\kappa_{\bar{s}}}_{\mathcal{R}_I^\kappa} \bar{p}'$ with $\bar{s}\ B_I\ \hat{\bar{p}}$. The $\kappa_{\bar{s}}$ action represent the synchronisation of all $\kappa_{\bar{s}[i]}$ actions, where $i \in I$. This means that $\forall \bar{q}, \bar{q}' \in \mathcal{S}_{\mathcal{R}_I^\kappa} : \bar{q} \xrightarrow{\kappa_{\bar{s}}}_{\mathcal{R}_I^\kappa} \bar{q}' \implies \bar{q} = \bar{s} \wedge \bar{q}' = \bar{s}$ (by Definition 6 and Definition 16). Thus, we have $\hat{\bar{p}} = \bar{s}$. It follows that $\bar{s}\ B_I\ \bar{s}$. □

In Chapter 3, we have shown that if a glue-state $s \in \mathcal{I}_{\mathcal{L}}$ is related to a state $s' \in \mathcal{S}_{\mathcal{R}}$, there exists a τ-path from s' to s (Lemma 3). A similar statement can be made for vector states in filtered pattern networks. Lemma 12 expresses that if a vector state $\bar{s} \in \mathcal{S}_{\mathcal{L}_I}$, of which some sub-states are glue-states, is related to $\bar{p} \in \mathcal{S}_{\mathcal{R}_I}$, then there exists a state $\hat{\bar{p}} \in \mathcal{S}_{\mathcal{R}_I}$ with the same glue-sub-states as $\bar{s}$ such that $\bar{p} \Rightarrow_{\mathcal{R}_I} \hat{\bar{p}}$ and $\hat{\bar{p}}$ is related to $\bar{s}$ as well.

Lemma 12. *Consider a vector of transformation rules $\bar{r}$ of size n given by a network transformation (Definition 12). Let $J \subseteq I$ and $I \subseteq 1..n$ be sets of indices, and let $\mathcal{L}_I$ and $\mathcal{R}_I$ be filtered left and right pattern networks produced from $\bar{r}$. Let B_I be a branching bisimulation relation such that $\mathcal{L}_I^\kappa \leftrightarrow_b \mathcal{R}_I^\kappa$, then*

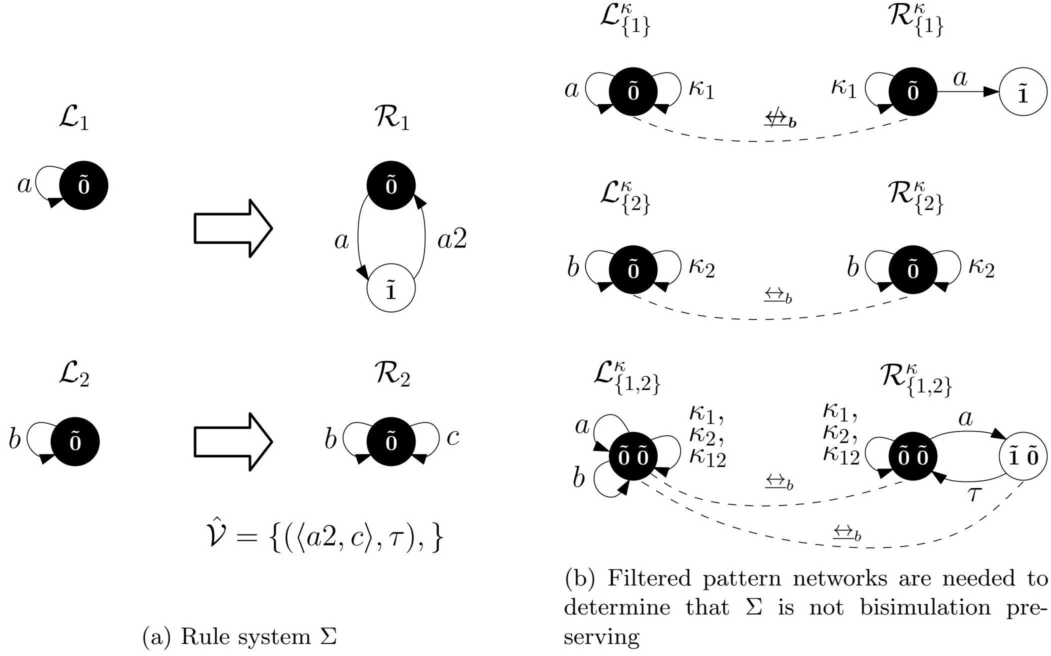
$$\forall \bar{s} \in \mathcal{S}_{\mathcal{L}_I}, \bar{p} \in \mathcal{S}_{\mathcal{R}_I} : \bar{s}\ B_I\ \bar{p} \wedge (\forall i \in J : \bar{s}[i] \in \mathcal{I}_{\mathcal{L}_i}) \implies$$
$$\exists \hat{\bar{p}} \in \mathcal{S}_{\mathcal{R}_I} : \bar{p} \Rightarrow_{\mathcal{R}_I} \hat{\bar{p}} \wedge \bar{s}\ B_I\ \hat{\bar{p}} \wedge \forall i \in J : \hat{\bar{p}}[i] = \bar{s}[i]$$

Proof. Let $\bar{s} \in \mathcal{S}_{\mathcal{L}_I}$ and $\bar{p} \in \mathcal{S}_{\mathcal{R}_I}$ such that $\bar{s}\ B_I\ \bar{p}$ and $\forall i \in J : \bar{s}[i] \in \mathcal{I}_{\mathcal{L}_i}$. Consider the κ-action $\kappa_{\bar{s}_J}$ produced from the synchronisation of the κ actions of all $\bar{s}[j]$ with $j \in J$. By Definition 16, it follows that $\bar{s} \in \mathcal{S}_{\mathcal{L}_I^\kappa}$, $\bar{p} \in \mathcal{S}_{\mathcal{R}_I^\kappa}$ and $\bar{s} \xrightarrow{\kappa_{\bar{s}_J}} \bar{s}$. Moreover, since $\bar{s}\ B_I\ \bar{p}$, we also have $\bar{p} \Rightarrow_{\mathcal{R}_I^\kappa} \hat{\bar{p}} \xrightarrow{\kappa_{\bar{s}_J}}_{\mathcal{R}_I^\kappa} \bar{p}'$ with $\bar{s}\ B_I\ \hat{\bar{p}}$. Since $\kappa_{\bar{s}_J}$ is unique for sub-states indexed by J we have $\forall i \in J : \hat{\bar{p}}[i] = \bar{s}[i]$. Finally, since τ is not the result of a κ-synchronisation, it follows that $\bar{p} \Rightarrow_{\mathcal{R}_I} \hat{\bar{p}}$. □

Handling inability to synchronise An LTS may want to perform a synchronising action in a certain state, but there is no guarantee that the synchronisation partners are able to perform the corresponding synchronisation action. Hence, when determining bisimulation preservation pattern networks that capture successful and unsuccessful synchronisation must be considered. As rule systems may be applied to any LTS network, all possible synchronisation scenarios must be considered.

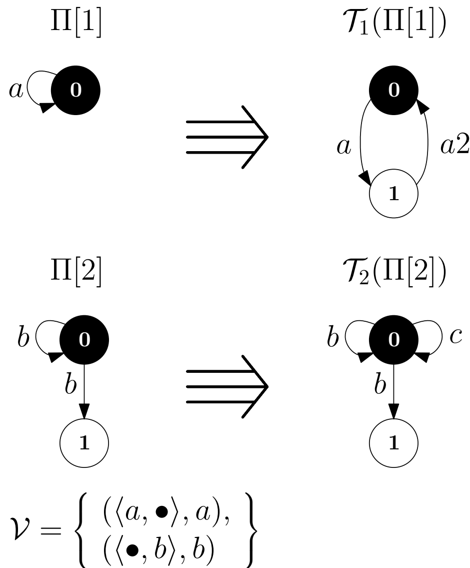
Figure 4.5 shows that unsuccessful synchronisation must be considered to properly detect that a given rule system does not preserve bisimulation. A rule system Σ that does not preserve

bisimulation is given in Figure 4.5a. 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 of the corresponding process LTS in Figure 4.5c. The bisimulation checks for the rule system Σ are presented in Figure 4.5b. Only when bisimulation is checked between $\mathcal{L}^{\kappa}_{\{1\}}$ and $\mathcal{R}^{\kappa}_{\{1\}}$ one can observe that the rule system is not bisimulation preserving. The κ_1, κ_2 transition represents two individual κ-transitions and the κ_{12} transition represents the synchronisation of the κ_1 and κ_2 actions. As indicated in Figure 4.5d, when the rule system is applied to the LTS network $\mathcal{M}$ in Figure 4.5c the resulting LTS network $\mathcal{T}_{\Sigma}(\mathcal{M})$ is not bisimilar to the original LTS network $\mathcal{M}$.

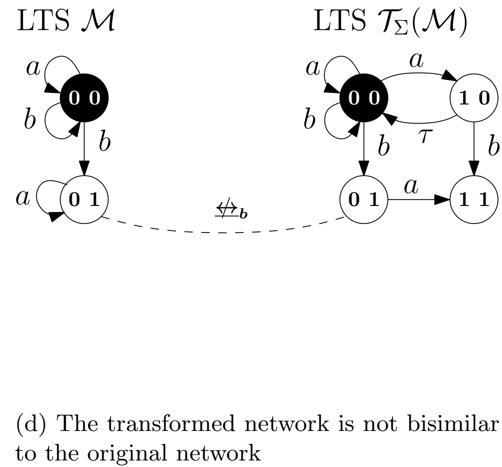


(a) Rule system Σ

(b) Filtered pattern networks are needed to determine that Σ is not bisimulation preserving

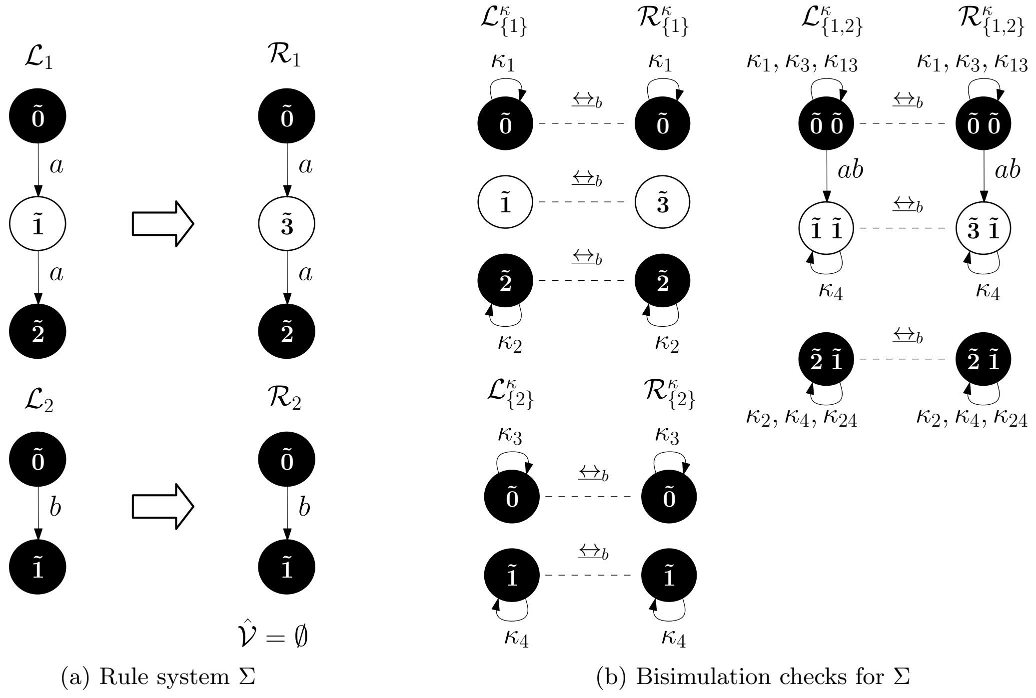


(c) Transforming an LTS network $\mathcal{M}$ using the rule system Σ



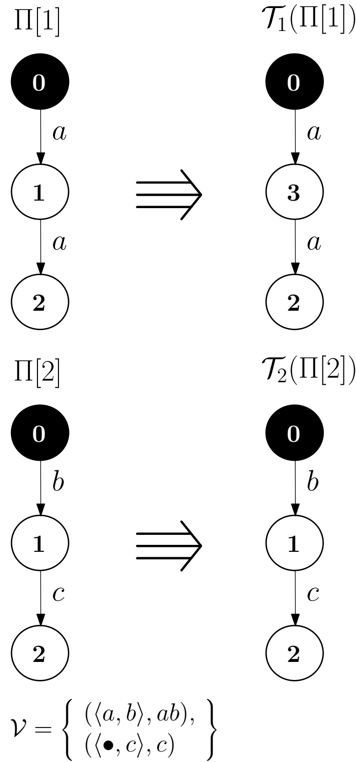
(d) The transformed network is not bisimilar to the original network

Figure 4.5: Both successful as unsuccessful synchronisation must be considered in checking for bisimulation preservation

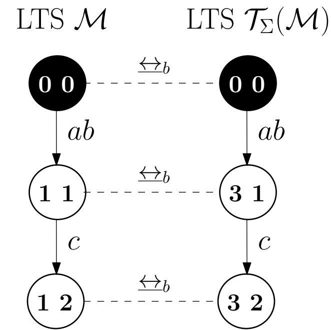


(a) Rule system Σ

(b) Bisimulation checks for Σ



(c) Transforming an LTS network $\mathcal{M}$ using the rule system Σ



(d) The system LTSs of $\mathcal{M}$ and $T_\Sigma(\mathcal{M})$ are bisimilar

Figure 4.6: A bisimulation preserving rule system applied onto an LTS network

4.4 Preservation of branching bisimulation

Reasoning about the generic bisimulation relation by example. To raise intuition of the contents of the general bisimulation relation for bisimulation preserving rule systems we will use an example which contains: (partly) matched vector states, successful synchronisation, and unsuccessful synchronisation.

Figure 4.6 shows the application of a rule system on an LTS network. Clearly, the rule system presented in Figure 4.6a preserves bisimulation: the first transformation rule simply relabels state $\tilde{1}$ to $\tilde{3}$, the second transformation rule does not change anything, and no new synchronisation laws are added. Figure 4.6b shows how the (filtered) pattern networks are related with respect to bisimulation. Rule system Σ is applied on an LTS network $\mathcal{M}$ as presented in Figure 4.6c. In Figure 4.6d shows the system LTSs before (LTS $\mathcal{M}$) and after (LTS $T_\Sigma(\mathcal{M})$) applying the rule system.

There are two ways to deduce that state $(0\ 0) \in \mathcal{S}_\mathcal{M} \cap \mathcal{S}_{T_\Sigma(\mathcal{M})}$ is related to itself: the state is not removed by the transformation, and the state $(\tilde{0}\ \tilde{0}) \in \mathcal{S}_{\mathcal{L}^\kappa_{\{1,2\}}}$ is related to itself since $(\tilde{0}\ \tilde{0}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{1,2\}}}$. Furthermore, since states $(1\ 1) \in \mathcal{S}_\mathcal{M}$ and $(3\ 1) \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ are fully matched and do not contain states matched by glue states their relation can only be derived from the relation between $\mathcal{L}^\kappa_{\{1,2\}}$ and $\mathcal{R}^\kappa_{\{1,2\}}$.

Relating states $(1\ 2) \in \mathcal{S}_\mathcal{M}$ and $(3\ 2) \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ is less trivial since the first element of the vector state is replaced and the second element of the vector state is not matched. There is no (filtered) pattern network which shows a trivial relation between these two vector states. Observe that the second elements of vector states $(\tilde{1}\ \tilde{1}) \in \mathcal{S}_{\mathcal{L}^\kappa_{\{1,2\}}}$ and $(\tilde{3}\ \tilde{1}) \in \mathcal{R}^\kappa_{\{1,2\}}$ are glue states. The κ_4-transition represents possible incoming and outgoing transitions to and from state $\tilde{1} \in \mathcal{S}_{\mathcal{L}^\kappa_{\{2\}}} \cap \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ that might not be present in the LTS pattern.

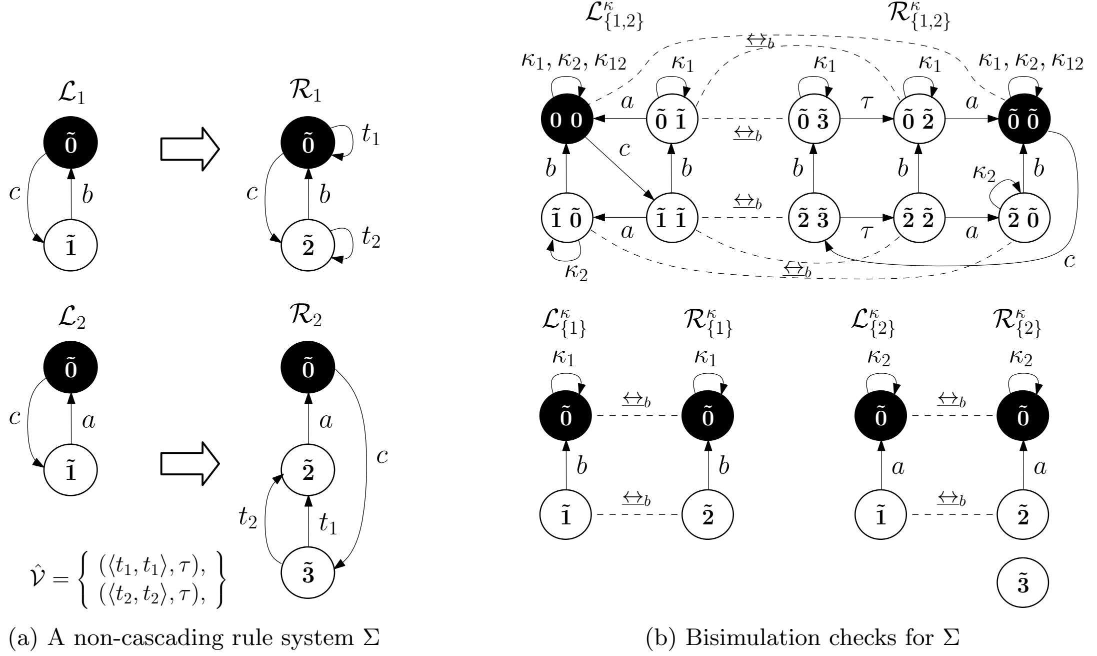
Vector state $(1\ 2) \in \mathcal{S}_\mathcal{M}$ behaves both as sub-state $(1) \in \mathcal{S}_1$ and sub-state $(2) \in \mathcal{S}_2$. Sub-state $(1) \in \mathcal{S}_1$ is replaced by the transformation (since $(1) \in m(\mathcal{L}_1 \setminus \mathcal{R}_1)$) and sub-state $(2) \in \mathcal{S}_2$ is not matched on, as a consequence, the two sub-states cannot perform synchronising actions. It can be said that the behaviour of $(1) \in \mathcal{S}_1$ and $(2) \in \mathcal{S}_2$ is *disjoint*. Hence, we can only relate states $(1\ 2) \in \mathcal{S}_\mathcal{M}$ and $(3\ 2) \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ iff there exists a bisimulation relation between $\mathcal{L}^\kappa_{\{1\}}$ and $\mathcal{R}^\kappa_{\{1\}}$ that relates sub-state $(\tilde{1}) \in \mathcal{S}_{\mathcal{L}^\kappa_{\{1\}}}$ and $(\tilde{3}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{1\}}}$. One could say that part of the bisimulation relation between $\mathcal{L}^\kappa_{\{1,2\}}$ and $\mathcal{R}^\kappa_{\{1,2\}}$ cascades into the bisimulation relation between $\mathcal{L}^\kappa_{\{1\}}$ and $\mathcal{R}^\kappa_{\{1\}}$. Rule systems for which this cascading effect holds in general are called *cascading rule systems*. The cascading rule system is formally defined in Definition 17. Cascading rule systems are a novel contribution to work of Wijs and Engelen [8, 24, 25].

Definition 17 (Cascading rule system)**.** *A rule system $\Sigma = (R, \hat{\mathcal{V}})$ is called a* cascading rule system*, iff for all sets of indices $I \subseteq 1..n$ and $J \subseteq I$ the following holds:*

$$\forall \bar{s} \in \mathcal{S}_{\mathcal{L}_I}, \bar{p} \in \mathcal{S}_{\mathcal{R}_I} : \bar{s}\ B_I\ \bar{p} \wedge (\forall i \in I \setminus J : \bar{s}[i] \in \mathcal{I}_{\mathcal{L}_i} \wedge \bar{s}[i] = \bar{p}[i]) \implies$$
$$(\forall \bar{s}' \in \mathcal{S}_{\mathcal{L}_J}, \bar{p}' \in \mathcal{S}_{\mathcal{R}_J} : (\forall i \in J : \bar{s}[i] = \bar{s}'[i] \wedge \bar{p}[i] = \bar{p}'[i]) \implies \bar{s}'\ B_J\ \bar{p}')$$

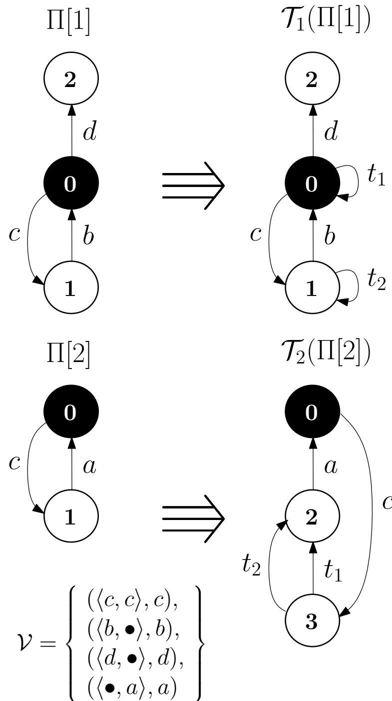
When a rule system is not cascading, bisimulation-preservation is not guaranteed for all inputs. Before arguing that this holds in general, we explain an illustrative example shown in Figure 4.7. A rule system Σ that is not cascading is presented in Figure 4.7a. The states are numbered such that matches can be identified by the state label, i.e. a state $\tilde{i}$ is matched onto state i of the corresponding process LTS in Figure 4.7c. The bisimulation checks for the rule system Σ are presented in Figure 4.7b. All bisimulation check fails to recognize that Σ is not bisimulation preserving. In the relation between $\mathcal{L}^\kappa_{\{1,2\}}$ and $\mathcal{R}^\kappa_{\{1,2\}}$ vector states $(\tilde{0}\ \tilde{1}) \in \mathcal{S}_{\mathcal{L}^\kappa_{\{1,2\}}}$ and $(\tilde{0}\ \tilde{3}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{1,2\}}}$ are related. However, sub-states $(\tilde{1}) \in \mathcal{S}_{\mathcal{L}^\kappa_{\{2\}}}$ and $(\tilde{3}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ are not related by the relation between $\mathcal{L}^\kappa_{\{2\}}$ and $\mathcal{R}^\kappa_{\{2\}}$. Therefore, rule system Σ is not cascading. The cause is the introduction of actions t_1 and t_2, which produce τ-transitions from sub-state $(\tilde{3}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ to sub-state $(\tilde{2}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ in such a way that sub-state $(\tilde{3}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ cannot perform any actions when pattern network $\mathcal{L}_{\{1,2\}}$

is left via sub-state $(\tilde{0}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{1\}}}$. However, sub-state $(\tilde{1}) \in \mathcal{S}_{\mathcal{R}^\kappa_{\{2\}}}$ can perform an a-action. Hence, as shown in Figure 4.7d, vector states $(3\ 1) \in \mathcal{S}_\mathcal{M}$ and $(3\ 3) \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ cannot be related. It follows that $\mathcal{M}$ and $T_\Sigma(\mathcal{M})$ are not bisimilar.

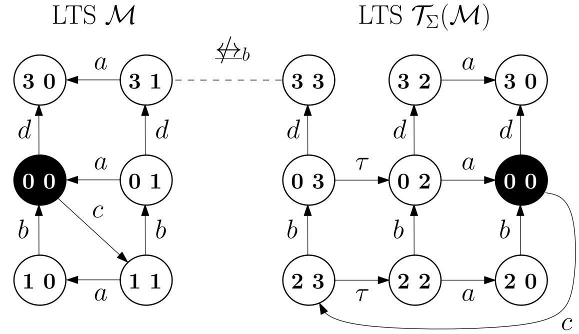


(a) A non-cascading rule system Σ

(b) Bisimulation checks for Σ



(c) Transforming an LTS network $\mathcal{M}$ using the rule system Σ



(d) The system LTS of $\mathcal{M}$ and the system LTS $T_\Sigma(\mathcal{M})$ are not bisimilar

Figure 4.7: The bisimulation checks for non-cascading rule system Σ do not recognize that Σ is not bisimulation-preserving for $\mathcal{M}$

We now argue that we cannot guarantee bisimulation-preservation for all inputs if a rule system is not cascading. Consider an LTS network $\mathcal{M} = (\Pi, \mathcal{V})$, a rule system $\Sigma = (R, \hat{\mathcal{V}})$, and two sets of indices $I \subseteq 1..n$ and $J \subseteq I$. Let $\bar{s} \in m(\mathcal{S}_{\mathcal{L}_I})$ and $\bar{p} \in \hat{m}(\mathcal{S}_{\mathcal{R}_I})$ be two bisimilar vector states with identical glue-states (indexed by $I \setminus J$). It is always possible to construct $\mathcal{M}$ such that there are transitions $\bar{s} \xrightarrow{\bar{t}\ a}_{\mathcal{M}} \bar{s}'$ and $\bar{p} \xrightarrow{\bar{t}\ a}_{T_\Sigma(\mathcal{M})} \bar{p}'$ with $Ac(\bar{t}) = I \setminus J$ and $\forall i \in I \setminus J : \bar{s}'[i] \notin m(\mathcal{S}_{\mathcal{L}_i})$, i.e. with the transitions the pattern networks $\mathcal{L}_I$ and $\mathcal{R}_I$ are left via sub-states shared by $\bar{s}$ and $\bar{p}$ that are matched on by glue-states. If $\mathcal{M}$ and $T_\Sigma(\mathcal{M})$ are bisimilar, then $\bar{s}'$ should be related to $\bar{p}'$. By Definition 8, we have $\forall i \in J : \bar{s}'[i] = \bar{s}[i]$. We refer to the sub-states of $\bar{s}'$ and $\bar{s}$ indexed by J as $\bar{s}_J$, and we refer to the sub-states of $\bar{p}$ indexed by J as $\bar{p}_J$. Similarly, we refer to the sub-states of $\bar{s}'$ indexed by I as $\bar{s}'_I$. If all sub-states in $\bar{s}_J$ are matched on by glue-states, it follows by Lemma 11 that the relation between $\bar{s}$ and $\bar{p}$ cascades to $\bar{s}_J$ and $\bar{p}_J$. If not all sub-states in $\bar{s}_J$ are matched on by glue-states, then there is a sub-state in $\bar{s}_J$ that is replaced by the transformation. Without loss of generality assume that all sub-states in $\bar{s}_J$ are replaced by the transformation. Hence, the behaviour of $\bar{s}'_I$ is disjoint from the behaviour of $\bar{s}'_J$. Assume that Σ is not cascading, i.e. the vector states that map on $\bar{s}_J$ and $\bar{p}_J$ are not bisimilar. Then, $\bar{s}'$ and $\bar{p}'$ are not bisimilar. As a consequence, $\mathcal{M}$ and $T_\Sigma(\mathcal{M})$ are not bisimilar. Therefore, bisimulation-preservation is not guaranteed for all inputs if a rule system is not cascading. We only consider cascading rule systems in the remainder of this thesis. An algorithm checking whether a rule system is cascading is a topic for future work.

The bisimulation-preservation check. Branching bisimilarity is a congruence for admissible LTS networks. Therefore, two pairs of pattern networks $\mathcal{L}_I$ and $\mathcal{R}_I$, $\mathcal{L}_J$ and $\mathcal{R}_J$, with $\mathcal{L}_I \leftrightarrow_b \mathcal{R}_I$ and $\mathcal{L}_J \leftrightarrow_b \mathcal{R}_J$, can be combined to form pattern networks $\mathcal{L}_{I\cup J}$ and $\mathcal{R}_{I\cup J}$ such that $\mathcal{L}_{I\cup J} \leftrightarrow_b \mathcal{R}_{I\cup J}$. By employing Lemma 13 we can avoid reasoning about dependency sets in the bisimulation relation that will be constructed.

Lemma 13. *Let $\Sigma = (R, \hat{\mathcal{V}})$ be a rule system. Let $\bar{r}$ be a vector of size n such that for all $i \in 1..n$, $\bar{r}[i] \in R$, then*

$$(\forall i \in 1..n, I \subseteq dep(i) : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I) \implies (\forall I \subseteq 1..n : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$$

Proof. By induction on n.

- Base case ($n = 1$). For all $i \in 1..n$ there is an $\bar{r}[i] \in R$. Hence, there is no $i \in 1..n$ that is not in the dependency set. It follows that for all $i \in 1..n$ we have $dep(i) = 1..n$.
- Step case with $(\forall i \in 1..n-1, I \subseteq dep(i) : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I) \implies (\forall I \subseteq 1..n-1 : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$ as Induction Hypothesis (IH). Given $(\forall i \in 1..n, I \subseteq dep(i) : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$, we also have $(\forall i \in 1..n-1, I \subseteq dep(i) : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$ and $\mathcal{L}^\kappa_n \leftrightarrow_b \mathcal{R}^\kappa_n$. By IH it follows that $(\forall I \subseteq 1..n-1 : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$. What remains to show is $(\forall I \subseteq 1..n-1 : \mathcal{L}^\kappa_{I\cup\{n\}} \leftrightarrow_b \mathcal{R}^\kappa_{I\cup\{n\}})$, i.e. the case with subsets containing n remains to be shown. Take an arbitrary $I \subseteq dep(i)$ with $i \in 1..n-1$, we have $\mathcal{L}^\kappa_n \leftrightarrow_b \mathcal{R}^\kappa_n$. By IH and congruence it follows that $\mathcal{L}^\kappa_{I\cup\{n\}} \leftrightarrow_b \mathcal{R}^\kappa_{I\cup\{n\}}$.

Hence, we have $(\forall i \in 1..n, I \subseteq dep(i) : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I) \implies (\forall I \subseteq 1..n : \mathcal{L}^\kappa_I \leftrightarrow_b \mathcal{R}^\kappa_I)$. □

Proposition 2 expresses that a rule system is branching-bisimulation-preserving if the κ-extended left- and right pattern networks of the dependency sets and their subsets are branching bisimilar. The bisimulation-preservation check follows directly from Proposition 2. Because of confluence of rule systems, exhaustive application of a rule system always results in the same final model. Let $\mathcal{M}_0$ and $\mathcal{M}_z$ be the original and final model respectively. By Proposition 2, each pair of intermediate models $\mathcal{M}_i$ and $\mathcal{M}_{i+1}$ are bisimilar. Because branching bisimulation is an equivalence relation, it follows that $\mathcal{M}_0$ is bisimilar to $\mathcal{M}_z$.

Proposition 2. *Let $\mathcal{M} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \hat{\mathcal{V}})$ be a cascading rule system. Let $\bar{r}$ be a vector of size n such that for all $i \in 1..n$, $\bar{r}[i] \in R$ with corresponding matches $m_i : \mathcal{S}_{\mathcal{L}} \to \mathcal{S}_{\Pi[i]}$ and $\hat{m}_i : \mathcal{S}_{\mathcal{R}} \to \mathcal{S}_{T(\Pi[i])}$. The following holds:*

$$(\forall i \in 1..n, I \subseteq dep(i) : \mathcal{L}_I^\kappa \leftrightarrow_b \mathcal{R}_I^\kappa) \implies \mathcal{M} \leftrightarrow_b T_\Sigma(\mathcal{M})$$

Proof. By Definition 2, of branching bisimilarity between LTSs, we have $\mathcal{M} \leftrightarrow_b T_\Sigma(\mathcal{M})$ iff there exists a branching bisimulation relation C such that $\mathcal{I}_{\mathcal{M}}\ C\ \mathcal{I}_{T_\Sigma(\mathcal{M})}$. By Lemma 13, we have $(\forall I \subseteq 1..n : \mathcal{L}_I^\kappa \leftrightarrow_b \mathcal{R}_I^\kappa)$. As a consequence, for any $I \subseteq 1..n$ there exists a branching bisimulation relation B_I with $\mathcal{I}_{\mathcal{L}_I^\kappa}\ B_I\ \mathcal{I}_{\mathcal{R}_I^\kappa}$. We define C as follows:

$$\begin{aligned} C = \{(\bar{s}, \bar{p}) \mid \forall i \in 1..n : \quad & (i \notin M(\bar{s}) \implies \bar{s}[i] \in \mathcal{S}_i \wedge \bar{s}[i] = \bar{p}[i]) \\ & \wedge (i \in M(\bar{s}) \implies \exists \bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}, \bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}} : \\ & \qquad \bar{s}_m\ B_{M(\bar{s})}\ \bar{p}_m \wedge \bar{s}_m \vdash_{M(\bar{s})} \bar{s} \wedge \bar{p}_m \vdash_{M(\bar{s})} \bar{p}) \} \end{aligned}$$

Consider an index $i \in 1..n$. The first case in the relation, $i \notin M(\bar{s})$, relates the sub-states of a state vector that are not matched by transformation rules. The second case in the relation, $i \in M(\bar{s})$, relates the matched sub-states of a state vector. We prove that C is a branching bisimulation relation, i.e. we show that Definition 2 holds for C. For readability, we emphasize conclusions that satisfy a given proof obligation.

In the proof we will sometimes construct state vectors from other state vectors. Let $\bar{s}$ and $\bar{p}$ be state vectors, and let $I \subseteq 1..n$ be a set of indices, we write $\bar{q} := \bar{s}[\bar{p}/I]$ to construct a state vector $\bar{q}$ with $\forall i \in 1..n : (i \in I \implies \bar{q}[i] = \bar{p}[i]) \wedge (i \notin I \implies \bar{q}[i] = \bar{s}[i])$.

- C relates the initial states of $\mathcal{M}$ and $T_\Sigma(\mathcal{M})$, i.e. $\mathcal{I}_{\mathcal{M}}\ C\ \mathcal{I}_{T_\Sigma(\mathcal{M})}$.

  Since $\mathcal{I}_{\Pi[i]} = \mathcal{I}_{T_i(\Pi[i])}$ we have $\mathcal{I}_{\mathcal{M}} = \mathcal{I}_{T_\Sigma(\mathcal{M})}$. Therefore, we only have to show that for all $\bar{s} \in \mathcal{I}_{\mathcal{M}}$ there exists a $\bar{p} \in \mathcal{I}_{\mathcal{M}}$ such that $\bar{s}\ C\ \bar{p}$. Let $\bar{s}$ be an arbitrary state in $\mathcal{I}_{\mathcal{M}}$, we can take $\bar{s}$ for $\bar{p}$ (as instance for the existential quantifier).

  What remains to be shown is that $\bar{s}\ C\ \bar{s}$. Consider an index $i \in 1..n$. Following relation C we distinguish two cases: $i \notin M(\bar{s})$ and $i \in M(\bar{s})$.

  - Case: $i \notin M(\bar{s})$. We must have $\boldsymbol{\bar{s}[i] = \bar{s}[i]}$, which is trivially true.
  - Case: $i \in M(\bar{s})$. We have to show there exist $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m\ B_{M(\bar{s})}\ \bar{p}_m$, $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, and $\bar{p}_m \vdash_{M(\bar{s})} \bar{s}$.
    By Lemma 9 we have a state $\boldsymbol{\bar{q} \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}}$ such that $\boldsymbol{\bar{q} \vdash_{M(\bar{s})} \bar{s}}$. Since the second condition of Definition 4 prevents the removal of initial states, we have $\forall i \in M(\bar{s}) : \bar{q}[i] \in \mathcal{I}_{\mathcal{L}_i}$, or simply $\bar{q} \in \mathcal{I}_{\mathcal{L}_{M(\bar{s})}}$ for short. From the lifted glue condition (Lemma 6) it follows that $\boldsymbol{\bar{q} \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}}$. Hence we can use $\bar{q}$ to instantiate the existential quantifier for both $\bar{s}_m$ and $\bar{p}_m$. By Lemma 11, we have $\boldsymbol{\bar{q}\ B_{M(\bar{s})}\ \bar{q}}$.

  In both cases we have $\boldsymbol{\bar{s}\ C\ \bar{s}}$.

- If $\bar{s}\ C\ \bar{p}$ and $\bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}'$ then either $a = \tau \wedge \bar{s}'\ C\ \bar{p}$, or $\bar{p} \Rightarrow_{T_\Sigma(\mathcal{M})} \hat{\bar{p}} \xrightarrow{a}_{T_\Sigma(\mathcal{M})} \bar{p}' \wedge \bar{s}\ C\ \hat{\bar{p}} \wedge \bar{s}'\ C\ \bar{p}'$.

  Let $(\bar{t}, a) \in \mathcal{V}$ be the synchronisation law that enables the transition $\bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}'$. We distinguish two cases: $M(\bar{s}) \cap Ac(\bar{t}) = \emptyset$ and $M(\bar{s}) \cap Ac(\bar{t}) \neq \emptyset$.

  1. Case: $M(\bar{s}) \cap Ac(\bar{t}) = \emptyset$. We have $\forall i \in Ac(\bar{t}) : i \notin M(\bar{s})$. We construct $\bar{p}' := \bar{p}[\bar{s}'/Ac(\bar{t})]$. By C we have $\forall i \in Ac(\bar{t}) : \bar{s}[i] = \bar{p}[i]$. It follows that $\boldsymbol{\bar{p} \xrightarrow{a}_{T_\Sigma(\mathcal{M})} \bar{p}'}$. Left to show is $\bar{s}'\ C\ \bar{p}'$. Let $i \in 1..n$:
     - Case: $i \notin M(\bar{s}')$. We have to show $\bar{s}'[i] = \bar{p}'[i]$.
       For $i \in Ac(\bar{t})$ we have $\bar{s}'[i] = \bar{p}'[i]$ by construction of $\bar{p}'$. For $i \in (1..n \setminus M(\bar{s})) \setminus Ac(\bar{t})$ we have $\bar{s}'[i] = \bar{s}[i] = \bar{p}[i] = \bar{p}'[i]$ (by Definition 8 and $\bar{s}C\bar{p}$). Hence, in both cases we have $\boldsymbol{\bar{s}'[i] = \bar{p}'[i]}$.

- Case: $i \in M(\bar{s}')$. We have to show there exist $\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}$ and $\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}$ such that $\bar{s}'_m \; B_{M(\bar{s}')} \; \bar{p}'_m$, $\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'$, and $\bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'$.
  We first establish some properties which will be needed to construct states bsa_m and $\bar{p}'_m$, and a relation with $\bar{s}'_m \; B_{M(\bar{s}')} \; \bar{p}'_m$. Since $\forall j \in M(\bar{s}') \cap Ac(\bar{t}) : j \in M(\bar{s}')$, by Lemma 9, there exists a $\bar{q}' \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}') \cap Ac(\bar{t})}}$ such that $\bar{q}' \vdash_{M(\bar{s}') \cap Ac(\bar{t})} \bar{s}'$. By applying Lemma 1 with $\bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}'$, and $\forall j \in Ac(\bar{t}) : j \notin M(\bar{s})$, it follows that $\forall j \in M(\bar{s}') \cap Ac(\bar{t}) : \bar{q}'[j] \in \mathcal{I}_{\mathcal{L}_j}$. Therefore, $\bar{q}' \in \mathcal{I}_{\mathcal{L}_{M(\bar{s}') \cap Ac(\bar{t})}}$.
  To relate $\bar{s}'$ and $\bar{p}'$ we need to find a relation $B_{M(\bar{s}')}$ relating two states that map on $\bar{s}'$ and $\bar{p}'$ respectively. If only active sub-states are matched we can use the property that initial states are related to themselves in $B_{M(\bar{s}')}$. In the opposite case, there is a $j \in 1..n \setminus Ac(\bar{t})$ and we can use $\bar{s} \; C \; \bar{p}$ to construct the required relation. We distinguish between the two mentioned cases:
  * Case: $\forall j \in 1..n \setminus Ac(\bar{t}), j \notin M(\bar{s}')$. Only sub-states of $\bar{s}'$ indexed by $Ac(\bar{t})$ are matched on. Hence, $M(\bar{s}') \cap Ac(\bar{t}) = M(\bar{s}')$. As $\bar{q}' \in \mathcal{I}_{\mathcal{L}_{M(\bar{s}') \cap Ac(\bar{t})}}$, by Lemma 6, we have $\boldsymbol{\bar{q}' \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$ and $\boldsymbol{\bar{q}' \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$. Since $\forall j \in Ac(\bar{t}) : \bar{s}'[j] = \bar{p}'[j]$ (by construction of $\bar{p}'$) and $\boldsymbol{\bar{q}' \vdash_{M(\bar{s}')} \bar{s}'}$, we also have $\boldsymbol{\bar{q}' \vdash_{M(\bar{s}')} \bar{p}'}$. Finally, by Lemma 11 it follows that $\boldsymbol{\bar{q}' \; B_{M(\bar{s}')} \; \bar{q}'}$.
  * Case: $\neg\forall j \in 1..n \setminus Ac(\bar{t}), j \notin M(\bar{s}')$. There is a matched sub-state of $\bar{s}'$ that is not indexed by $Ac(\bar{t})$, i.e. there exists a $j \in 1..n \setminus Ac(\bar{t})$ such that $j \in M(\bar{s}')$. Since $j \notin Ac(\bar{t})$ we have $j \in M(\bar{s})$ (by Definition 8). Since $\bar{s} \; C \; \bar{p}$ and $j \in M(\bar{s})$ it follows that there exist $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m \; B_{M(\bar{s})} \; \bar{p}_m$, $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, and $\bar{p}_m \vdash_{M(\bar{s})} \bar{p}$. Furthermore, by Lemma 11 we have $\bar{q}' \; B_{M(\bar{s}')} \; \bar{q}'$. Let $\bar{s}'_m := \bar{s}_m[\bar{q}'/Ac(\bar{t})]$ and $\bar{p}'_m := \bar{p}_m[\bar{q}'/Ac(\bar{t})]$, applying congruence to $\bar{q}' \; B_{M(\bar{s}')} \; \bar{q}'$ and $\bar{s}_m \; B_{M(\bar{s})} \; \bar{p}_m$ we get $\bar{s}'_m \; B_{(M(\bar{s}') \cap Ac(\bar{t})) \cup M(\bar{s})} \; \bar{p}'_m$. Since $\forall k \in 1..n \setminus Ac(\bar{t}) : k \in M(\bar{s}) \iff k \in M(\bar{s}')$ (by Definition 8) and $M(\bar{s}) \cap Ac(\bar{t}) = \emptyset$, we have $(M(\bar{s}') \cap Ac(\bar{t})) \cup M(\bar{s}) = M(\bar{s}')$. Hence, $\boldsymbol{\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$, $\boldsymbol{\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}}$, and $\boldsymbol{\bar{s}'_m \; B_{M(\bar{s}')} \; \bar{p}'_m}$. Finally, since $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, $\bar{p}_m \vdash_{M(\bar{s})} \bar{p}$, and $\bar{q}' \vdash_{M(\bar{s}') \cap Ac(\bar{t})} \bar{s}'$, by construction of $\bar{s}'_m$ and $\bar{p}'_m$, it follows that $\boldsymbol{\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'}$ and $\boldsymbol{\bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'}$.

In all cases we have $\boldsymbol{\bar{s}' \; C \; \bar{p}'}$.

2. Case: $M(\bar{s}) \cap Ac(\bar{t}) \neq \emptyset$. We distinguish whether there exists a matching transition:

(a) Case: $\exists i \in Ac(\bar{t}), q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_i q'$. Part of transition $\bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}'$ is matched on. By C we have states $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m \; B_{M(\bar{s})} \; \bar{p}_m$, $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, and $\bar{p}_m \vdash_{M(\bar{s})} \bar{p}$. By Lemma 10 there exists $\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ such that $\bar{s}'_m \vdash_{M(\bar{s})} \bar{s}'$ and there is a transition $\bar{s}_m \xrightarrow{\bar{t} \; a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$. Since $\bar{s}_m \; B_{M(\bar{s})} \; \bar{p}_m$, by Definition 2, we have the following two cases:

   i. Case: $a = \tau$ with $\bar{s}'_m \; B_{M(\bar{s})} \; \bar{p}_m$. We have to show that $\bar{s}' \; C \; \bar{p}$. Let $i \in 1..n$:
   - Case: $i \notin M(\bar{s}')$. We have to show $\bar{s}'[i] = \bar{p}'[i]$.
     Since $\bar{s}_m \xrightarrow{\bar{t} \; a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$ and $i \notin M(\bar{s}')$, by Definition 8, it follows that $i \notin Ac(\bar{t})$. Therefore, we have $\bar{s}'[i] = \bar{s}[i]$ and $i \notin M(\bar{s})$ (both by Definition 8). Furthermore, from $i \notin M(\bar{s})$ and $\bar{s} \; C \; \bar{p}$ it follows that $\bar{s}[i] = \bar{p}[i]$. Consequently, we have $\boldsymbol{\bar{s}'[i] = \bar{p}'[i]}$.
   - Case: $i \in M(\bar{s}')$. We have to show there exist $\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}$ and $\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}$ such that $\bar{s}'_m \; B_{M(\bar{s}')} \; \bar{p}_m$, $\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'$, and $\bar{p}_m \vdash_{M(\bar{s}')} \bar{p}$.
     Since $\bar{s}_m \xrightarrow{\bar{t} \; a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$, it follows that $\forall i \in Ac(\bar{t}) : i \in M(\bar{s}) \wedge i \in M(\bar{s}')$. By Definition 8, we have $\forall i \in 1..n \setminus Ac(\bar{t}) : i \in M(\bar{s}) \iff i \in M(\bar{s}')$. Hence, $M(\bar{s}) = M(\bar{s}')$. Consequently, we have $\boldsymbol{\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$ and $\boldsymbol{\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}}$ with $\boldsymbol{\bar{s}'_m \; B_{M(\bar{s}')} \; \bar{p}_m \wedge \bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}' \wedge \bar{p}_m \vdash_{M(\bar{s}')} \bar{p}}$.

   In both cases we have $\boldsymbol{\bar{s}' \; C \; \bar{p}}$.

ii. Case: $\bar{p}_m \Rightarrow_{\mathcal{R}_{M(\bar{s})}} \hat{\bar{p}}_m \xrightarrow{a}_{\mathcal{R}_{M(\bar{s})}} \bar{p}'_m$ with $\bar{s}_m\ B_{M(\bar{s})}\ \hat{\bar{p}}_m$ and $\bar{s}'_m\ B_{M(\bar{s})}\ \bar{p}'_m$. We have to show $\exists \hat{p}, \bar{p}' \in \mathcal{S}_{T_\Sigma(\mathcal{M})} : \bar{p} \Rightarrow_{T_\Sigma(\mathcal{M})} \hat{\bar{p}} \xrightarrow{a}_{T_\Sigma(\mathcal{M})} \bar{p}' \wedge \bar{s}\ C\ \hat{\bar{p}} \wedge \bar{s}'\ C\ \bar{p}'$.
By Lemma 8 there is a $\boldsymbol{\hat{\bar{p}} \in \mathcal{S}_{T_\Sigma(\mathcal{M})}}$ such that $\boldsymbol{\bar{p} \Rightarrow_{T_\Sigma(\mathcal{M})} \hat{\bar{p}}}$, $\boldsymbol{\hat{\bar{p}}_m \vdash_{M(\bar{s})} \hat{\bar{p}}}$, and $\forall j \in 1..n \setminus M(\bar{s}) : \hat{\bar{p}}[j] = \bar{p}[j]$. By Lemma 7, there is a $\boldsymbol{\bar{p}' \in \mathcal{S}_{T_\Sigma(\mathcal{M})}}$ such that $\boldsymbol{\hat{\bar{p}} \xrightarrow{a}_{T_\Sigma(\mathcal{M})} \bar{p}'_m}$, $\bar{p}'_m \vdash_{M(\bar{s})} \bar{p}'$, and $\forall j \in 1..n \setminus M(\bar{s}) : \bar{p}'[j] = \hat{\bar{p}}[j]$. Hence, the transition requirements are satisfied. Moreover, by Lemma 14 we have $\boldsymbol{\bar{s}\ C\ \hat{\bar{p}}}$. What remains to be shown is $\bar{s}'\ C\ \bar{p}'$. Let $i \in 1..n$:

- Case: $i \notin M(\bar{s}')$. We have to show $\bar{s}'[i] = \bar{p}'[i]$.
  Since $\bar{s}_m \xrightarrow{\bar{t}\ a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$ and $i \notin M(\bar{s}')$, by Definition 8, it follows that $i \notin Ac(\bar{t})$. Therefore, we have $\bar{s}'[i] = \bar{s}[i]$ and $i \notin M(\bar{s})$ (both by Definition 8). Moreover, from $i \notin M(\bar{s})$ and $\bar{s}\ C\ \bar{p}$ it follows that $\bar{s}[i] = \bar{p}[i]$. Since $\bar{p}[i] = \hat{\bar{p}}[i] = \bar{p}'[i]$ (by construction of $\hat{\bar{p}}$ and $\bar{p}'$), we have $\boldsymbol{\bar{s}'[i] = \bar{p}'[i]}$.
- Case: $i \in M(\bar{s}')$. We have to show there exist $\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}$ and $\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}$ such that $\bar{s}'_m\ B_{M(\bar{s}')}\ \bar{p}'_m$, $\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'$, and $\bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'$.
  As $\bar{s}_m \xrightarrow{\bar{t}\ a}_{\mathcal{L}_{M(\bar{s})}} \bar{s}'_m$, it follows that $\forall i \in Ac(\bar{t}) : i \in M(\bar{s}) \wedge i \in M(\bar{s}')$. Furthermore, by Definition 8, $\forall i \in 1..n \setminus Ac(\bar{t}) : i \in M(\bar{s}) \iff i \in M(\bar{s}')$. Hence, $M(\bar{s}) = M(\bar{s}')$. Consequently, we have $\boldsymbol{\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$ and $\boldsymbol{\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}}$ with $\boldsymbol{\bar{s}'_m\ B_{M(\bar{s}')}\ \bar{p}'_m \wedge \bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}' \wedge \bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'}$.

In both cases we have $\boldsymbol{\bar{s}'\ C\ \bar{p}'}$.

(b) Case: $\forall i \in Ac(\bar{t}) : \neg(\exists q, q' \in \mathcal{S}_{\mathcal{L}_i} : m_i(q) = \bar{s}[i] \wedge m_i(q') = \bar{s}'[i] \wedge q \xrightarrow{\bar{t}[i]}_i q')$. There is no match on transition $\bar{s} \xrightarrow{a}_{\mathcal{M}} \bar{s}'$.
We have $M(\bar{s}) \cap Ac(\bar{t}) \neq \emptyset$, it follows that there is a $k \in M(\bar{s}) \cap Ac(\bar{t})$. Hence, by $\bar{s}\ C\ \bar{p}$, there exist $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\bar{p}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, $\bar{p}_m \vdash_{M(\bar{s})} \bar{p}$, and $\bar{s}_m\ B_{M(\bar{s})}\ \bar{p}_m$. For all $j \in Ac(\bar{t})$ there is no matching transition or there is no state $q \in \mathcal{S}_{\mathcal{L}_j}$ such that $m_j(q) = \bar{s}'[j]$. Hence, by Lemma 1, it follows that $\forall j \in M(\bar{s}) \cap Ac(\bar{t}) : \bar{s}_m[j] \in \mathcal{I}_{\mathcal{L}_j}$. By Lemma 12, there exists a $\hat{\bar{p}}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{p}_m \Rightarrow_{\mathcal{R}_{M(\bar{s})}} \hat{\bar{p}}_m$, $\bar{s}_m\ B_{M(\bar{s})}\ \hat{\bar{p}}_m$, and $\forall j \in M(\bar{s}) \cap Ac(\bar{t}) : \hat{\bar{p}}_m[j] = \bar{s}_m[j]$. Moreover, by Lemma 8 there is a $\boldsymbol{\hat{\bar{p}} \in \mathcal{S}_{T_\Sigma(\mathcal{M})}}$ such that $\boldsymbol{\hat{\bar{p}}_m \vdash_{M(\bar{s})} \hat{\bar{p}}}$, $\boldsymbol{\bar{p} \Rightarrow_{T_\Sigma(\mathcal{M})} \hat{\bar{p}}}$, and $\forall j \in 1..n \setminus M(\bar{s}) : \hat{\bar{p}}[j] = \bar{p}[j]$. Let $\bar{p}' := \hat{\bar{p}}[\bar{s}'/Ac(\bar{t})]$, by construction, we have $\boldsymbol{\bar{p}' \in \mathcal{S}_{T_\Sigma(\mathcal{M})}}$. Since $\forall j \in M(\bar{s}) \cap Ac(\bar{t}) : \hat{\bar{p}}_m[j] = \bar{s}_m[j]$, by injectivity of m_j (Definition 4), it follows that $\forall j \in Ac(\bar{t}) : \hat{\bar{p}}[j] = \bar{s}[j]$. Therefore, by construction of $\bar{p}'$, we have $\boldsymbol{\hat{\bar{p}} \xrightarrow{a}_{T_\Sigma(\mathcal{M})} \bar{p}'}$. Moreover, by Lemma 14 we have $\boldsymbol{\bar{s}\ C\ \hat{\bar{p}}}$. What remains to be shown is $\bar{s}'\ C\ \bar{p}'$. Let $i \in 1..n$:

- Case: $i \notin M(\bar{s}')$. We have to show $\bar{s}'[i] = \bar{p}'[i]$.
  For $i \in Ac(\bar{t})$ we have $\bar{s}'[i] = \bar{p}'[i]$ by construction of $\bar{p}'$. Furthermore, for $i \in (1..n \setminus M(\bar{s})) \setminus Ac(\bar{t})$ we have $\bar{s}'[i] = \bar{s}[i] = \bar{p}[i] = \hat{\bar{p}}[i] = \bar{p}'[i]$ (by Definition 8, $\bar{s}\ C\ \bar{p}$, and construction of $\hat{\bar{p}}$ and $\bar{p}'$). Hence, in both cases we have $\boldsymbol{\bar{s}'[i] = \bar{p}'[i]}$.
- Case: $i \in M(\bar{s}')$. We have to show there exist $\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}$ and $\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}$ such that $\bar{s}'_m\ B_{M(\bar{s}')}\ \bar{p}'_m$, $\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'$, and $\bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'$.
  By Definition 8 only sub-states index by $Ac(\bar{t})$ change. Sub-states index by $Ac(\bar{t})$ may transition from a matched sub-state to another matched sub-state, or such a matched sub-state may transition to a sub-state that is not matched (or vice versa). We will construct states $\bar{s}' \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}$ and $\bar{p}' \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}$ by considering the two disjoint sets $M(\bar{s}') \setminus Ac(\bar{t})$ and $M(\bar{s}') \cap Ac(\bar{t})$, such that we can relate $\bar{s}'$ and $\bar{p}'$ using congruence.
  First, we focus on $M(\bar{s}') \setminus Ac(\bar{t})$. Consider the earlier established state $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$. In order to use Definition 17 we split $\bar{s}_m$ and $\hat{\bar{p}}_m$ each in two parts: $\bar{s}_{mI} \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}) \cap Ac(\bar{t})}}$ and $\bar{s}_{mJ} \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}) \setminus Ac(\bar{t})}}$, and $\hat{\bar{p}}_{mI} \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}) \cap Ac(\bar{t})}}$ and $\hat{\bar{p}}_{mJ} \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}) \setminus Ac(\bar{t})}}$. By Lemma 1, it follows that $\forall j \in M(\bar{s}) \cap Ac(\bar{t}) : \bar{s}_m[j] \in \mathcal{I}_{\mathcal{L}_j}$. Hence, we have $\bar{s}_{mI} \in \mathcal{I}_{\mathcal{L}_{M(\bar{s}) \cap Ac(\bar{t})}}$. Since $\forall j \in M(\bar{s}) \cap Ac(\bar{t}) : \hat{\bar{p}}_m[j] = \bar{s}_m[j]$, we have

$\bar{s}_{mI} = \hat{\bar{p}}_{mI}$. By Definition 17, it follows that $\bar{s}_{mJ}\ B_{M(\bar{s})\setminus Ac(\bar{t})}\ \hat{\bar{p}}_{mJ}$. Finally, since $M(\bar{s}') \setminus Ac(\bar{t}) = M(\bar{s}) \setminus Ac(\bar{t})$, we get $\bar{s}_{mJ}\ B_{M(\bar{s}')\setminus Ac(\bar{t})}\ \hat{\bar{p}}_{mJ}$.
Next, we prove some properties for the set $M(\bar{s}') \cap Ac(\bar{t})$. By Lemma 9, there exists a state $\bar{q}' \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')\cap Ac(\bar{t})}}$ such that $\bar{q}' \vdash_{M(\bar{s}')\cap Ac(\bar{t})} \bar{s}'$. By Lemma 1, we have $\bar{s}_{mI} \in \mathcal{I}_{\mathcal{L}_{M(\bar{s})\cap Ac(\bar{t})}}$. Hence, we also have $\bar{q}' \vdash \bar{p}'$. Furthermore, from Lemma 11, it follows that $\bar{q}'\ B_{M(\bar{s}')\cap Ac(\bar{t})}\ \bar{q}'$.
Let $\bar{s}'_m := \bar{q}'[\bar{s}_{mJ}/M(\bar{s}') \setminus Ac(\bar{t})]$ and $\bar{p}'_m := \bar{q}'[\hat{\bar{p}}_{mJ}/M(\bar{s}') \setminus Ac(\bar{t})]$. By construction of $\bar{s}'_m$ and $\bar{p}'_m$, we have $\boldsymbol{\bar{s}'_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s}')}}}$, $\boldsymbol{\bar{p}'_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s}')}}}$, $\boldsymbol{\bar{s}'_m \vdash_{M(\bar{s}')} \bar{s}'}$, and $\boldsymbol{\bar{p}'_m \vdash_{M(\bar{s}')} \bar{p}'}$. Finally, by congruence, it follows that $\boldsymbol{\bar{s}'_m\ B_{M(\bar{s}')}\ \bar{p}'_m}$.

In both cases we have $\boldsymbol{\bar{s}'\ C\ \bar{p}'}$.

- If $\bar{s}\ C\ \bar{p}$ and $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ then either $a = \tau \wedge \bar{s}\ C\ \bar{p}'$, or $\bar{s} \Rightarrow_{\mathcal{M}} \hat{\bar{s}} \xrightarrow{a}_{\mathcal{M}} \bar{s}' \wedge \hat{\bar{s}}\ C\ \bar{p} \wedge \bar{s}'\ C\ \bar{p}'$.

  This case has not been mechanically verified. However, this case is symmetrical to the previous case with one exception: $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ is enabled by some $(\bar{t}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. Therefore, when transition $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ is not matched on, we have to show that $(\bar{t}, a) \in \mathcal{V}$.

  Let $\bar{p}, \bar{p}' \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ such that $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ is enabled by some $(\bar{t}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. We consider the cases where transition $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ is not matched on. Assume for a contradiction that $(\bar{t}, a) \in \hat{\mathcal{V}}$. By condition 5 of Definition 13, all actions $\bar{t}[i]$ with $i \in 1..n$ are not actions of original process $\Pi[i]$, i.e. we have $\forall i \in 1..n : \bar{t}[i] \notin \mathcal{A}_i$. Hence, these actions must be introduced by $\mathcal{R}_i$. It follows that $\forall i \in 1..n : \bar{t}[i] \in \mathcal{A}_{\mathcal{R}_i} \setminus \mathcal{A}_{\mathcal{L}_i}$. However, this means that transition $\bar{p} \xrightarrow{a}_{T(\mathcal{M})} \bar{p}'$ must be matched on, which contradicts our assumption that the transition is not matched. Hence, we must have $(\bar{t}, a) \in \mathcal{V}$.

□

In the proof of Proposition 2, there are two cases where we construct a state $\hat{\bar{p}} \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ from two related states $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ and $\bar{p} \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ such that the matched parts of $\bar{s}$ and $\hat{\bar{p}}$ are related. The fact that $\bar{s}$ and $\hat{\bar{p}}$ are related by C follows almost directly from the definition of C, as shown in Lemma 14.

Lemma 14. *Consider a vector of transformation rules $\bar{r}$ of size n given by a network transformation (Definition 12). Let $\mathcal{L}_{M(\bar{s})}$ and $\mathcal{R}_{M(\bar{s})}$ be filtered left and right pattern networks produced from $\bar{r}$. Let $B_{M(\bar{s})}$ be a branching bisimulation relation such that $\mathcal{L}^{\kappa}_{M(\bar{s})} \leftrightarrow_b \mathcal{R}^{\kappa}_{M(\bar{s})}$, then*

$$\begin{array}{l}\forall \bar{s} \in \mathcal{S}_{\mathcal{M}}, \bar{p}, \hat{\bar{p}} \in \mathcal{S}_{T_\Sigma(\mathcal{M})} : \bar{s}\ C\ \bar{p} \wedge \forall j \in 1..n \setminus M(\bar{s}) : \hat{\bar{p}}[j] = \bar{p}[j] \wedge \\ \quad \begin{pmatrix} \exists \bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}, \bar{p}_m, \hat{\bar{p}}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}} : \bar{p}_m \vdash_{M(\bar{s})} \bar{p} \wedge \hat{\bar{p}}_m \vdash_{M(\bar{s})} \hat{\bar{p}} \\ \wedge\ \bar{s}_m \vdash_{M(\bar{s})} \bar{s} \wedge \bar{s}_m\ B_{M(\bar{s})}\ \hat{\bar{p}}_m \wedge \end{pmatrix} \implies \bar{s}\ C\ \hat{\bar{p}}\end{array}$$

Proof. Let $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ and $\bar{p} \in \mathcal{S}_{T_\Sigma(\mathcal{M})}$ such that $\bar{s}\ C\ \bar{p}$. Furthermore, let $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\bar{p}_m, \hat{\bar{p}}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, $\bar{p}_m \vdash_{M(\bar{s})} \bar{p}$, $\hat{\bar{p}}_m \vdash_{M(\bar{s})} \hat{\bar{p}}$, and $\bar{s}_m\ B_{M(\bar{s})}\ \hat{\bar{p}}_m$. Additionally, assume that $\forall j \in 1..n \setminus M(\bar{s}) : \hat{\bar{p}}[j] = \bar{p}[j]$ by construction of $\hat{\bar{p}}$. We show that $\bar{s}\ C\ \hat{\bar{p}}$. Let $i \in 1..n$. By definition of C, we distinguish two cases:

- Case: $i \notin M(\bar{s})$. We have to show $\bar{s}[i] = \hat{\bar{p}}[i]$.

  By $\bar{s}\ C\ \bar{p}$ and construction of $\hat{\bar{p}}$, it follows that $\bar{s}[i] = \bar{p}[i] = \hat{\bar{p}}[i]$.

- Case: $i \in M(\bar{s})$. We have to show there exist $\bar{s}_m \in \mathcal{S}_{\mathcal{L}_{M(\bar{s})}}$ and $\hat{\bar{p}}_m \in \mathcal{S}_{\mathcal{R}_{M(\bar{s})}}$ such that $\bar{s}_m\ B_{M(\bar{s})}\ \hat{\bar{p}}_m$, $\bar{s}_m \vdash_{M(\bar{s})} \bar{s}$, and $\hat{\bar{p}}_m \vdash_{M(\bar{s})} \hat{\bar{p}}$, this is trivially satisfied by the premises of this lemma.

In both cases we have $\bar{s}\ C\ \hat{\bar{p}}$. □

Chapter 5

Conclusion and Future Work

This work formally verified the theory proposed by Wijs and Engelen [8, 25] using the Coq interactive theorem prover. We have formally proven the correctness of the bisimulation-preservation check with respect to the application of a transformation rule to a single process LTS. We have improved upon some of the definitions making the theory's notions more consistent. Furthermore, we propose two novel contributions to the theory proposed by Wijs and Engelen in order to prove the correctness of the bisimulation-preservation check. These contributions are κ-synchronisation laws and cascading rule systems.

We have shown that κ-synchronisation laws are required to take into account synchronising transitions that enter and leave a pattern network. Furthermore, when a rule system is not cascading, it is possible to construct an LTS network $\mathcal{M}$ such that the original network $\mathcal{M}$ and the transformed network $T_{\Sigma}(\mathcal{M})$ are not bisimilar while the check gives a positive result. Hence, to guarantee that a transformation preserves bisimulation for all possible inputs, the rule system must be cascading. Finally, we have proven the correctness of the bisimulation-preservation check with respect to the application of a transformation on an LTS network. For this proof we constructed a generic relation that relates original and transformed LTS networks and have shown that this relation is a bisimulation. The proof is formally verified using an interactive theorem prover.

The bisimulation-preservation check is limited input models that are admissible with respect to τ-transitions. In addition, the check is limited to rule systems that are confluent, cascading and synchronously uniform. However, it can be checked whether models and rule system satisfy these limitations. Even when a transformation does not preserve bisimulation for a given property, it may still be possible that said property holds for the output model of a specific instance of the transformation. Nevertheless, transformations that do preserve bisimulation can be reused without the need for additional verification.

Future work. In this thesis we have established the correctness of the bisimulation-preservation check with respect to branching bisimulation. However, the original check [8, 25] uses divergence-sensitive-branching bisimulation. In the future, divergence-sensitivity can be added to our bisimulation definition in Coq. This extension will only add proof obligations for the divergence-sensitivity condition, i.e. we will have to prove that divergence of states is preserved given that the two LTS patterns of a κ-extended transformation rule are divergence-sensitive branching bisimilar. Additionally, Wijs and Engelen [23] proposed a weakening of the synchronisation uniformity pre-condition and an extension of their approach which efficiently handles transformations which leave the communication interface intact. The weakening and extension can be added to the formalization in Coq. Weakening the pre-conditions in the formalization would only require reproving a few lemma's. Formalization of the extension would require the construction of a new generic bisimulation relation between original and transformed LTS network to reason about transformation which leaves the communication interface intact.

At the moment of the writing of this thesis, the authors of REFINER [26] have introduced κ-synchronisation in the property-preservation check. However, the implementation does not

yet check whether rule systems are cascading (Definition 17). We want to develop an efficient algorithm which checks whether a rule system is a cascading rule system. Consider a rule system $\Sigma = (R, \hat{\mathcal{V}})$, where $1..n$ are indices uniquely identifying $r \in R$. Let $I \subseteq 1..n$ and $J \subseteq I$ be two sets of indices. Moreover, let B_I and B_J be bisimulation relations such that $\mathcal{L}_I^\kappa \underline{\leftrightarrow}_b \mathcal{R}_I^\kappa$ and $\mathcal{L}_J^\kappa \underline{\leftrightarrow}_b \mathcal{R}_J^\kappa$ respectively. We can construct a relation $\hat{B}_J$ between $\mathcal{L}_J^\kappa$ and $\mathcal{R}_J^\kappa$ from bisimulation relations B_I and B_J such that entries from B_I cascade to $\hat{B}_J$ according to Definition 17. A trivial approach to check whether Σ is cascading is to check for all $I \subseteq 1..n$ and $J \subseteq I$ whether $\hat{B}_J$ is a bisimulation relation. However, this approach performs a bisimulation check twice for each J and may not scale well with parallel processing. We conjecture that it is possible to construct $\hat{B}_J$ from a bisimulation relation $B_{\{1..n\}}$ between $\mathcal{L}_{\{1..n\}}^\kappa$ and $\mathcal{R}_{\{1..n\}}^\kappa$. Should this be possible, then we only have to perform one bisimulation check for each $J \subset 1..n$. These checks can be performed in parallel.

To further increase the reliability of the property-preservation check approach, we could encode the corresponding algorithm in Coq as a proof. An implementation of the algorithm that is correct-by-construction can then be generated from the Coq proof.

As another area of future work, we would like to investigate how the property-preservation check can be applied on modelling and transformation languages. It would be interesting to investigate what class of modelling and transformation languages can be semantically formalized such that the property-preservation check can be applied to verify transformations. Furthermore, the current implementation of the property-preservation check approach does not return counter-examples when a transformation is deemed not property-preserving. To guide developers in developing transformations, it is vital to generate counter-examples for model transformations that do not preserve a given property. Such counter-examples help the developers understand *why* their transformation may not preserve a given property.

Bibliography

[1] L. Ab. Rahim and J. Whittle. A survey of approaches for verifying model transformations. *Software and Systems Modeling*, pages 1–26, 2013. 7, 9

[2] C. Baier and J.-P. Katoen. *Principles of model checking.* MIT Press, 2008. 1, 2, 4, 11, 12

[3] P. Baldan, A. Corradini, H. Ehrig, R. Heckel, and B. König. Bisimilarity and Behaviour-preserving Reconfigurations of Open Petri Nets. In *Proceedings of the 2nd International Conference on Algebra and Coalgebra in Computer Science (CALCO'07)*, pages 126–142. Springer-Verlag, 2007. 4, 9

[4] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In *Proceedings of the Integration of Model Driven Development and Model Driven Testing (IMDDMDT) workshop at the 2nd European Conference on Modelling Foundations and Applications (ECMDA'06)*, July 2006. 2

[5] B. W. Boehm and P. N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, 1988. 1

[6] S.M.J. de Putter. Formalizing proofs of bisimulation-preserving graph transformations in Coq. Seminar formal system analysis at Einhoven University of Technology, 2013. Available at: `http://1drv.ms/1cMz2Wg`. 2, 5

[7] H. Ehrig, F. Hermann, and C. Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *Electronic Communications of the EASST*, 18, 2009. 7

[8] L. J. P. Engelen and A. J. Wijs. Checking Property Preservation of Refining Transformations for Model-Driven Development. CS-Report 12-08, Eindhoven University of Technology, 2012. 2, 4, 11, 22, 23, 24, 29, 32, 39

[9] M. García and R. Möller. Certification of Transformation Algorithms in Model-Driven Software Development. In Bleek W.-G., J. Raasch, and H. Züllighoven, editors, *Software Engineering*, volume 105 of *Lecture Notes in Informatics*, pages 107–118. Gesellschaft für Informatik, 2007. 9

[10] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner. Towards Verified Model Transformations. In *Proceedings of the 3rd International Workshop on Model Development, Validation and Verification (MoDeVVa'06)*, pages 78–93. Le Commissariat à l'Energie Atomique - CEA, 2006. 8

[11] G. Goos. Compiler Verification and Compiler Architecture. *Electronic Notes in Theoretical Computer Science*, 65(2):1, 2002. 2

[12] F. Lang. Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In J. Romijn, G. Smith, and J. van de Pol, editors, *Proceedings of the 5th International Conference on Integrated Formal Methods (iFM'05)*, volume 3771 of *Lecture Notes in Computer Science*, pages 70–88. Springer, 2005. 4, 19, 20

[13] K. Lano and S. Kolahdouz-Rahimi. Specification and Verification of Model Transformations Using UML-RSDS. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM'10)*, Lecture Notes in Computer Science, pages 199–214. Springer, 2010. 8

[14] L. Lúcio and H. Vangheluwe. Symbolic execution for the verification of model transformations. Technical Report SOCS-TR-2013, McGill University, 2013. Available at: `http://msdl.cs.mcgill.ca/people/levi/files/MTSymbExec.pdf`. 9

[15] E. M. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In Reinhard Wilhelm, editor, *Informatics*, Lecture Notes in Computer Science, pages 176–194. Springer Berlin Heidelberg, 2001. 2

[16] R. Mateescu and A. J. Wijs. Property-Dependent Reductions for the Modal Mu-Calculus. In *Proceedings of the 18th International SPIN Workshop on Model Checking of Software (SPIN'11)*, volume 6823 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 2011. 4

[17] I. Poernomo and J. Terrell. Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq. In *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM'10)*, pages 56–73. Springer, 2010. 8

[18] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001. 1

[19] H. J. Schneider and H. Ehrig. Grammars on Partial Graphs. *Acta Informatica*, 6(3):297–316, 1976. 14

[20] K. Stenzel, N. Moebius, and W. Reif. Formal Verification of QVT Transformations for Code Generation. In J. Whittle, T. Clark, and T. Kühne, editors, *In Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MODLS'11)*, volume 6981 of *Lecture Notes in Computer Science*, pages 533–547. Springer, 2011. 8

[21] The Institute of Electrical and Eletronics Engineers. IEEE Standard Glossary of Software Engineering Terminology. IEEE Standard, September 1990. 3

[22] R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, May 1996. 9, 12

[23] A. J. Wijs. Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics. In *Proceedings of the 10th International Symposium on Formal Aspects of Component Software (FACS'13)*, volume 8348 of *Lecture Notes in Computer Science*. Springer, 2013. To appear. 4, 39

[24] A. J. Wijs and L. J. P. Engelen. Incremental Formal Verification for Model Refining. In *Proceedings of the 9th Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa'12)*, pages 29–34. ACM, 2012. 2, 4, 22, 23, 29, 32

[25] A. J. Wijs and L. J. P. Engelen. Efficient Property Preservation Checking of Model Refinements. In N. Piterman and S. A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, volume 7795 of *Lecture Notes in Computer Science*, pages 565–579. Springer, 2013. 2, 4, 21, 22, 23, 29, 32, 39

[26] A. J. Wijs and L. J. P. Engelen. REFINER: Towards Formal Verification of Model Transformations. In *Proceedings of the 6th NASA Formal Methods Symposium (NFM'14)*, volume 8430 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2014. 4, 21, 39

Appendix A

Coq in a nutshell

The Coq proof assistant[1] is an interactive theorem prover. For a good understanding of how the proofs presented in this paper relate to the presented Coq listings, it is essential to know the basics of Coq.

Typing and verification In Coq mathematical definitions, executable algorithms and theorems can be expressed in a formal language called Calculus of Inductive Construction (CIC). CIC is based on typed λ-calculus, a (logical) formula is a type and the corresponding λ-term is a proof of the formula. Coq mechanically verifies proofs by checking whether a λ-term indeed has the given type, i.e. Coq checks whether a given proof indeed proves the corresponding formula using a type-checking algorithm.

In Coq every object has a type. That a given object `x` has type `T` is denoted by `x : T`. Types have a type called a sort whenever a type is manipulated as a term. The set of sorts $\mathcal{S}$ is defined as

$$\mathcal{S} = \{\texttt{Set}, \texttt{Prop}, \texttt{Type}(i) \mid i \in \mathbb{N}\}$$

where `Set : Type(1)`, `Prop : Type(1)` and $\texttt{Type}(i)$ `:` $\texttt{Type}(i+1)$ for all $i \in \mathbb{N}$ as shown in Figure A.1.

The sort `Prop` represents the type of logical propositions. The sort `Set` represents the type of small sets or universes; e.g booleans, naturals, and function types over these data types. Unlike in set theory, set operations are not available on objects of type `Set`. The sort $\texttt{Type}(i+1)$, like `Set` contains small sets, but also large sets such as `Set`, `Prop`, and $\texttt{Type}(i)$. Furthermore, $\texttt{Type}(i+1)$ contains all products, subsets and function types over the sorts in $\texttt{Type}(i)$ as well.

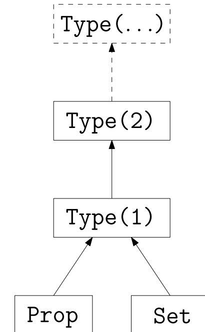


Figure A.1: The CIC typing hierarchy

Coq syntax Consider a function `F : A → B → C`, we have $A \rightarrow B \rightarrow C = A \rightarrow (B \rightarrow C)$ since types are right associative. Terms are left associative, we have `F a b = (F a) b`. Moreover, since `F a : B → C` we have `F a b : C`. Let P and Q be logical propositions, and let x and y be instances of some type T. Table A.1 shows a summary of the syntax for logical propositions. The syntax of the universal and existential quantifiers is presented in Table A.2.

Table A.1: Coq syntax for logical propositions

$\perp$	$\top$	$x = y$	$x \neq y$	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \rightarrow Q$	$P \Leftrightarrow Q$
`False`	`True`	`x = y`	`x <> y`	`~P`	`P /\Q`	`P \/ Q`	`P -> Q`	`P <-> Q`

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

Table A.2: Coq syntax for logical quantifiers

$\forall x \in T : P$	`forall x:T, P`
$\exists x \in T : P$	`exists x:T, P`

Definitions and proofs There are a few constructs which are used to specify definitions and proofs. Definitions can be specified by means of the keywords `Definition` and `Record`. Listing A.1 defines a predicate of type `(A → B) → Prop` on a function `F : A → B` stating that `F` is injective: function `F` is injective iff `Injective F` holds. Note that the type of the variables quantified in the `forall` statement are left implicit in this example, Coq is able to derive types for variables and definitions in most situations.

```
Definition Injective (F : A -> B) : Prop :=
  forall a1 a2, F a1 = F a2 -> a1 = a2.
```

Listing A.1: Injective predicate for a function

We can also formalize an injective function in a `Record`. The definition presented in Listing A.2 states that `InjectiveFunction` is a function `F : A → B` for which `Injective_cond` holds. The `:>` symbols after `F` declare `F` as a coercion from `InjectiveFunction` to `A → B`, this means that an object `IF : InjectiveFunction` can be used as a function of type `A → B`.

```
Record InjectiveFunction := {
  F :> A -> B;
  Injective_cond : forall a1 a2, F a1 = F a2 -> a1 = a2
}.
```

Listing A.2: Record defining an injective function

A `Record` can contain multiple objects, for example a tuple with an `x : X` and a `y : Y` can be defined as shown in Listing A.3. The `x` and `y` elements can be extracted from a record object `t : TupleXY` using `(x t)` and `(y t)` respectively.

```
Record TupleXY := {
  x : X;
  y : Y
}.
```

Listing A.3: A tuple with an `x : X` and a `y : Y`

Proofs for propositions can be constructed using the `Lemma` or `Theorem` keywords. The distinction between `Lemma` and `Theorem` is only syntactical, i.e. `Lemma` and `Theorem` can be used interchangeably. During the construction of a proof Coq shows the hypotheses available and the goals (proof obligations). When all goals are solved a proof is completed with `Qed`, which validates and records the proof. A Coq proof for the commutativity of the conjunction operator is shown in Listing A.4. Several tactics are applied to prove the goal `(A ∧ B) → (B ∧ A)`. First, `intro` introduces the `(A ∧ B)` part before the implication as a hypothesis `H`, the goal becomes `(B ∧ A)`. The `destruct` tactic, splits hypothesis `H` into two hypotheses. Finally `split ; assumption` proves the goal. This final step first uses `split` to split the goal in the two sub-goals `A` and `B`, after which both goals are proven by our hypotheses using `assumption`. For other tactics and constructs used in this work, please refer to the Coq reference manual[2].

```
Lemma AndCommutative : (A /\ B) -> (B /\ A).
Proof.
intro H.
destruct H.
split ; assumption.
Qed.
```

Listing A.4: Proof that the conjunction operator is commutative

[2] http://coq.inria.fr/refman/

Appendix B

Formalization

Section B.1 discusses formalizations with respect to a single process LTS. Section B.2 is dedicated to the formalization of the bisimulation-preservation check with respect to the transformation of multiple, concurrent process LTSs. The Coq formalization has been checked with version 8.4pl2 of the Coq IDE. The verification code is available at: http://1drv.ms/1s3cE1A. The formal proofs in Coq mostly follow the same structure as the proofs presented in this thesis. One proof-step in the presented proofs usually corresponds to several, much smaller, steps in Coq.

B.1 Transformation of a single process

An LTS is a tuple $\langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G} \rangle$ as defined in Definition 1. We formalize this definition in Coq as shown in Listing B.1. A set of states or actions is defined by means of the `Ensemble` type. A set of vector states with length n, called `States`, is denoted as `States : Ensemble (State n)` and is of type `State n` $\rightarrow$ `Prop`. A state `s : State` is a member of the set `States : Ensemble State` iff `States s` holds. The set of actions and the set of initial states are defined similarly. Given an action `a : Act` and two states `s : State`, `s' : State`, an LTS `G` has transition `s` $\xrightarrow{\mathtt{a}}_{\mathtt{G}}$ `s'` iff `Trans G a s s'` holds.

```
Record LTS := {
  States : Ensemble (State n); (* States 𝒮 (vector size n) *)
  Acts : Ensemble Act;      (* Actions 𝒜*)
  Trans : Act -> (State n) -> (State n) -> Prop; (* Transitions 𝒯*)
  Init : Ensemble State;    (* Initial states ℐ *)

  (* 𝒯 ⊆ 𝒜 × 𝒮 × 𝒮 *)
  trans_cond : forall a s s', Trans a s s' -> States s /\ States s' /\ Acts a;
  (* ℐ ⊆ 𝒮 *)
  init_cond : forall s : State, Init s -> States s;
  (* ∅ ⊂ ℐ *)
  init_not_empty_cond : exists s, Init s;
  (* τ ∈ 𝒜 *)
  tau_cond : Acts tau
}.
```

Listing B.1: Formalization of an LTS in Coq

The formalization of Definition 2 (branching bisimulation) in Coq is presented in Listing B.2. The binary relation B is formalized as the relation `BR` on the set of all states. For ease of use, bisimulation-relation `BR` is declared as coercion from `BB` to the relation `State` $\rightarrow$ `State` $\rightarrow$ `Prop`. The `B_states` property states that relation `BB G1 G2` relates the states of LTSs `G1` and `G2`. Bisimulation between the two LTSs is expressed by the `B_init`. The transfer conditions of branching bisimulation are expressed in `B_branching` and `B_branching_sym`. The `TauPath` function, defined by `clos_refl_trans State (Trans G tau)`, represents the reflexive transitive closure of the τ-transition, i.e. for states `t`, `th` and LTS `G` we have `t` $\Rightarrow_{\mathtt{G}}$ `th` iff `TauPath G t th` holds.

```
Record BB (G1 : LTS)(G2 : LTS) := {
  BR :> State n -> State n -> Prop; (* Relation B *)

  (* B relates states of G1 to states of G2 *)
  B_states : forall s t, BR s t -> States G1 s /\ States G2 t;

  (* The initial states of both LTSs must be related *)
  B_init : forall s, (Init G1 s -> exists t, Init G2 t /\ BR s t)
                  /\ (Init G2 s -> exists t, Init G1 t /\ BR s t);

  (* B is a branching bisimulation *)
  B_branching :
    forall s t, BR s t ->              (* for all s,t such that s B t *)
      forall a s', Trans G1 a s s' -> (* if s -a->G1 s' *)
         (* either a = τ ∧ s' B t *)
         (a = tau /\ BR s' t)
         (* or t =>G2 t̂ -a-> t' with s B t̂ and s' B t' *)
      \/ exists th t', (TauPath G2 t th /\ Trans G2 a th t' /\ BR s th /\ BR s' t')
         ;

  B_branching_sym :
    forall s t, BR s t ->              (* for all s,t such that s B t *)
      forall a t', Trans G2 a t t' -> (* if t -a->G2 t' *)
         (a = tau /\ BR s t')
      \/ exists sh s', (TauPath G1 s sh /\ Trans G1 a sh s' /\ BR sh t /\ BR s' t')
}.
```

Listing B.2: Formalization of branching bisimulation (Definition 1) in Coq

A transformation rule r, described in Definition 3, consists of a left LTS pattern $\mathcal{L}$ and a right LTS pattern $\mathcal{R}$ such that $\mathcal{I}_\mathcal{L} = \mathcal{I}_\mathcal{R} = \mathcal{S}_\mathcal{L} \cap \mathcal{S}_\mathcal{R}$. We formalize the notion of a transformation rule in Coq as shown in Listing B.3. The `L` and `R` fields represent the left and right LTS patterns respectively. The `glue_cond` property states that the set of initial states (the glue-states) of both LTS patterns is comprised of the intersection of the set of states of the left and right LTS patterns, i.e. `Init L = Init R = (States L) ∩ (States R)`.

```
Record TransRule := {
  L : LTS; (* Left pattern ℒ *)
  R : LTS; (* Right pattern ℛ *)

  (* ℐ_ℒ = ℐ_ℛ = 𝒮_ℒ ∩ 𝒮_ℛ *)
  glue_cond : Init L = Init R /\ Init L = (IntersectS (States L) (States R))
}.
```

Listing B.3: Formalization of a transformation rule (Definition 3) in Coq

Listing B.4 presents the formalization of match $m : \mathcal{S}_\mathcal{H} \to \mathcal{S}_\mathcal{G}$ of the left LTS pattern $\mathcal{H}$ of a transformation rule on a graph $\mathcal{G}$ in Coq (Definition 4). Since a match is a partial function with respect to the set of all states, we define the match function `m` as a relation such that two states `s` and `s'` are related iff `(m s s')` holds. All states $s \in \mathcal{S}_\mathcal{H}$ are mapped by match `m` as indicated by `m_matches`. The `m_states` property expresses that if $m(q) = s$, we have $q \in \mathcal{S}_\mathcal{H}$ and $s \in \mathcal{S}_\mathcal{G}$, this corresponds to $m : \mathcal{S}_\mathcal{L} \to \mathcal{S}_\mathcal{G}$. The second condition of a match is represented by `m_trans`, it states that there must be matching transitions for all transitions in $\mathcal{T}_\mathcal{H}$. The `m_init` property states that initial states may only be matched on by glue-states, this corresponds to the third condition of the definition of a match. Furthermore, `D_cond` corresponds to the dangling conditions (the fourth condition of a match).

```
Record Match (H : LTS)(G : LTS) := {
  m :> State 1 -> State 1 -> Prop;

  (* A match maps all states in H*)
  m_matches : forall p, States H p <-> exists s, m p s;
```

```
  (* A match maps states in H to states in G, i.e. m : H -> G  *)
  m_states : forall p s, m p s -> States H p /\ States G s;

  (* m is injective *)
  m_injective : forall s1 s2 s', (m s' s1 /\ m s' s2)
                                 \/ (m s1 s' /\ m s2 s') -> s1 = s2;

  (* A non-glue state may not match onto an initial state, i.e.
      ∀i ∈ I_G, s ∈ S_H : m(s) = i ⟹ s ∈ I_H  *)
  m_init : forall i, Init G i -> forall s, m s i -> States (L r) s /\ States (R r)
      s;

  (* The transition labels must match, i.e. ∀p1 -a->_H p2 : m(p1) -a->_G m(p2) *)
  m_trans : forall a p1 p2, Trans H a p1 p2 ->
              exists s1 s2, m p1 s1 /\ m p2 s2 /\ Trans G a s1 s2;

  (* Dangling conditions *)
  D_cond : forall s1 s2 p1 a,
        States H p1 /\ ~Init H p1 (* p1 ∈ S_H \ I_H  *)
     /\ States G s2 (* s2 ∈ G  *)
    -> ( (* (D1) m(p1) -a->_G s2 ⟹ ∃p2 ∈ S_H : p1 -a->_H p2 ∧ m(p2) = s2  *)
          (m p1 s1 /\ Trans G a s1 s2 ->
            exists p2, States H p2 /\ Trans H a p1 p2 /\ m p2 s2)
          (* (D2) s2 -a->_G m(p1) ⟹ ∃p2 ∈ S_H : p2 -a->_H p1 ∧ m(p2) = s2  *)
       /\ (m p1 s1 /\ Trans G a s2 s1 ->
            exists p2, States H p2 /\ Trans H a p2 p1 /\ m p2 s2)
    )
}.
```

Listing B.4: Formalization of a transformation rule's match (Definition 4) in Coq

The formalization of Lemma 1 is presented in Listing B.5. The lemma expresses that whenever a transition is not matched but a state of the transition is matched, then the matched state must be a glue-state.

```
Lemma D_cond_reverse : forall H G (mr : Match H G) s1 s2 p a, (* H, G, m : S_H -> S_G  *)
  (* s1 -a->_G s2 ⟹  *)
  Trans G a s1 s2 -> ( (* m(p) = s1 ∧ ∀p2 ∈ S_H : m(p2) ≠ s2 ∨ ¬p -a->_H p2  *)
                       (mr p s1 /\ (forall p2, ~mr p2 s2 \/ ~Trans H a p p2))
                      (* (m(p) = s2 ∧ ∀p1 ∈ S_H : m(p1) ≠ s1 ∨ ¬p1 -a->_H p)) *)
                    \/ (mr p s2 /\ (forall p1, ~mr p1 s1 \/ ~Trans H a p1 p)))
                    -> Init H p. (* p ∈ I_H  *)
```

Listing B.5: Formalization of Lemma 1 in Coq

Listing B.6 presents the formalization of the κ-extended transformation rule, this is an applied version of Definition 6. The KFun function produces a unique κ_s action for a state s. The produced action cannot be τ as expressed by kfun_not_tau.

The record TransRuleK defines the κ-extension of some $\mathcal{L}$ and $\mathcal{R}$ of a transformation rule $r = (\mathcal{L}, \mathcal{R})$ given a KFun function that produces κ-actions. The kacts_cond condition ensures that κ-actions are not already present in $\mathcal{A}_\mathcal{L}$ and $\mathcal{A}_\mathcal{R}$. The a_cond property adds κ-actions to $\mathcal{A}_\mathcal{L}$ and $\mathcal{A}_\mathcal{R}$ to produce $\mathcal{A}_{\mathcal{L}^\kappa}$ and $\mathcal{A}_{\mathcal{R}^\kappa}$. The sets $\mathcal{T}_{\mathcal{L}^\kappa}$ and $\mathcal{T}_{\mathcal{R}^\kappa}$ are defined by t_cond. A transition is in $\mathcal{T}_{\mathcal{L}^\kappa}$ iff the transition was already present in $\mathcal{T}_\mathcal{L}$ or the transition is the κ-self-loop of a glue-state, i.e. $\mathcal{T}_{\mathcal{L}^\kappa}$ is $\mathcal{T}_\mathcal{L}$ with additional κ-self-loops from glue-states. The set $\mathcal{T}_{\mathcal{L}^\kappa}$ is defined similarly.

```
(* A function that produces a unique action per state *)
Record KFun := {
  kfun :> State 1 -> Act;

  kfun_injective : forall s s', kfun s = kfun s' -> s = s';
  kfun_not_tau : forall s, kfun s <> tau
}.

(* The κ-extended transformation rule *)
Record TransRuleK (r : TransRule)(k_fun : KFun) := {
```

```
  ker :> TransRule; (* κ-extension of r *)

  (* κ-actions not in A_L and A_R *)
  kact_cond : forall s, ~Acts (L r) (k_fun s) /\ ~Acts (R r) (k_fun s);

  (* S_{L^κ} = S_L ∧ S_{R^κ} = S_R *)
  s_cond : States (L ker) = States (L r) /\ States (R ker) = States (R r);
  (* I_{L^κ} = I_L ∧ I_{R^κ} = I_R *)
  i_cond : Init (L ker) = Init (L r) /\ Init (R ker) = Init (R r);
  (* A_{L^κ} = A_L ∪ {κ_s | s ∈ I_L} ∧ A_{R^κ} = A_R ∪ {κ_s | s ∈ I_R} *)
  a_cond : forall a, (Acts (L ker) a -> Acts (L r) a
                      \/ exists s, Init (L r) s /\ k_fun s = a)
                  /\ (Acts (R ker) a -> Acts (R r) a
                      \/ exists s, Init (R r) s /\ k_fun s = a)
                  /\ (Acts (L r) a -> Acts (L ker) a)
                  /\ (Acts (R r) a -> Acts (R ker) a);
  (* κ-self-loops are added *)
  t_cond : forall a s s',
       (Trans (L ker) a s s' <-> Trans (L r) a s s'
          \/ (k_fun s = a /\ s = s' /\ Init (L r) s))
    /\ (Trans (R ker) a s s' <-> Trans (R r) a s s'
          \/ (k_fun s = a /\ s = s' /\ Init (R r) s))
}.
```

Listing B.6: Formalization of the κ-extension of a transformation rule (Definition 6) in Coq

The formalization of Definition 5 is given in Listing B.7. The `Transform` record declares LTS `TG`, corresponding to LTS $T(\mathcal{G})$, as a coercion from `Transform` to `LTS`. LTS `TG` has the following properties: `IT_cond`, `ST_cond`, `TT_cond`,and `AT_cond`. These four properties correspond to $\mathcal{I}_\mathcal{G}$, $\mathcal{S}_m$, $\mathcal{A}_m$, and $\mathcal{T}_m$ respectively. Most notable is the definition of $\mathcal{T}_m$, it is defined as $\mathcal{T}_\mathcal{G}$ removing the transitions in $\mathcal{T}_\mathcal{L}$ and adding transitions in $\mathcal{T}_\mathcal{R}$.

```
(* Transformation rule r, LTS G, and match m : S_L → S_G *)
Record Transform {r : TransRule}{G : LTS}(mL : Match (L r) G) := {
  TG :> LTS; (* T(G) *)

  mR : Match (R r) TG; (* Match m̂ : S_R → S_{T(G)} *)
  mR_cond : forall q, Init (L r) q -> mL q = mR q;

  (* Initial state are identical to I_G *)
  IT_cond : Init TG = Init G;

  (* S_m = (S_G \ m(S_L)) ∪ m̂(S_R) *)
  ST_cond :
      let mR_States := image_mr mR (States (R r)) in
      let mL_States := image_mr mL (States (L r)) in
        States TG = UnionS (SetminusS (States G) mL_States) mR_States;

  (* T_m = (T_G \ {m(s) -a->_G m(s') | s -a->_L s'}) ∪ {m̂(s) -a->_{T(G)} m̂(s') | s -a->_R s'} *)
  TT_cond :
      forall a s s',
        Trans TG a s s' <-> (
             (* T_G \ {m(s) -a->_G m(s') | s -a->_L s'} *)
             (Trans G a s s' /\ ~(exists q q', mL q s /\ mL q' s'
                                 /\ Trans (L r) a q q'))
          \/ (* {m̂(s) -a->_{T(G)} m̂(s') | s -a->_R s'} *)
             exists q q', mR q s /\ mR q' s' /\ Trans (R r) a q q'
         );

  (* A_m = {a | ∃s -a-> s' ∈ T_m} ∪ {τ} *)
  AT_cond : forall a, Acts TG a <-> (a = tau \/ exists s s', Trans TG a s s')
}.
```

Listing B.7: Formalization of LTS transformation (Definition 5) in Coq

B.2 Transformation of multiple synchronizing processes

Given in Listing B.8 is the Coq formalization of the LTS Network (Definition 7). The `SyncLaw` record defines a synchronisation law. Say we have a synchronisation law `v` corresponding to $v = (\bar{t}, a)$, then `sync v` represents the synchronisation vector $\bar{t}$ and `act v` represents the result action a. The synchronisation vector `sync` is formalized as a list of length n. Consider a list `l` of length `n`, `nth i l d` function returns its third argument `d` if `i >= n`. Hence, we have `nth i (sync v) None = None` for $\bar{t}[i] = \bullet$ and we have `nth i (sync v) None = Some b` for $\bar{t}[i] = b$.

The LTS network of some size n is formalized by the `LTSNetwork` record. The list of n processes is represented by `Procs`. The set of synchronisation laws $\mathcal{V}$ corresponds to `SyncN`. For all synchronisation laws $(\bar{t}, a)$ we must have $\forall i \in 1..n : \bar{t}[i] \in \mathcal{A}_i \cup \{\bullet\}$, this property is expressed by `NoUnknownSyncActs` and enforced by the `SyncActs_cond` condition. Moreover, the `Adm_cond` property expresses that the LTS networks is admissible.

```
(* Syncronisation law (t̄,a) *)
Record SyncLaw {n} := {
  sync : listn (option Act) n; (* Synchronisation vector t̄
                                  • is represented by None *)
  act  : Act                   (* Result action a *)
}.

(* All actions in a synchronisation vector are members of the corresponding
   process LTS, i.e. ∀(t̄,a) ∈ 𝒱, i ∈ Ac(t̄) : t̄[i] ∈ 𝒜_i ∪ {•} *)
Definition NoUnknownSyncActs {n}(V : Ensemble (@SyncLaw n))(Glist : list LTS) :=
  forall v, V v ->                                  (* ∀(t̄,a) ∈ 𝒱 *)
    forall i a ,(nth i (sync v) None) = Some a -> (* ∀i ∈ Ac(t̄) *)
       Acts (nth i Glist d_lts) a.                  (* t̄[i] ∈ 𝒜_i *)

(* LTS network *)
Record LTSNetwork {n} := {
  Procs : listn (@LTS 1) n;        (* Vector of LTSs Π (with state size 1) *)
  SyncN : Ensemble (@SyncLaw n); (* Set of synchronisation rules 𝒱*)

  SyncActs_cond : NoUnknownSyncActs SyncN Procs;
  Adm_cond      : Admissible SyncN Procs (* The LTS network is admissible *)
}.
```

Listing B.8: Formalization of LTS network (Definition 7) in Coq

The formalization of the system LTS of an LTS network is presented in Listing B.9. The set of all vector states $\mathcal{S}_\mathcal{M}$ is formalized by `NetStates`. The set of actions that may be performed by the system LTS is given by `NetActs`. The `NetTransSL` definition formalizes the transition with a given synchronisation law $\bar{s} \xrightarrow{\bar{t}\ a}_\mathcal{M} \bar{s}'$. An a-transition ($\bar{s} \xrightarrow{a}_\mathcal{M} \bar{s}'$)in the system LTS is described by `NetTrans`. The initial states of the system LTS are formalized by `NetInit`. Finally, the system LTS of an LTS network `M` is formally constructed by `SystemLTS`. Since `(SystemLTS M)` is of type `LTS`, we may use `(SystemLTS M)` as an LTS in any function or definition operating on LTSs; e.g `Acts (SystemLTS M)` will return the set `NetActs M`.

```
(* 𝒮_𝓜 = 𝒮_1 × ··· × 𝒮_n *)
Definition NetStates {n}(M : @LTSNetwork n)(s : State n) :=
  forall i, i < n -> States (nth i (Procs M) d_lts) (nthS i s).

(* 𝒜_𝓜 = {a | (t̄,a) ∈ 𝒱} ∪ {τ} *)
Definition NetActs {n}(M : @LTSNetwork n)(a : Act) :=
  a = tau \/ exists v, (SyncN M) v /\ act v = a.

(* Transition with known sync law, i.e. s̄ -t̄ a-> s̄' *)
Definition NetTransSL {n}(M : @LTSNetwork n)(v : SyncLaw)(s1 s2 : State n) :=
  SyncN M v /\
    forall (i : nat), i < n -> (* ∀i ∈ 1..n *)
      match nth i (sync v) None with
```

```
          (* t̄[i] = • ∧ s̄[i] = s̄′[i] ∧ s̄[i] ∈ S_i  *)
        | None =>     nth i s1 d_str = nth i s2 d_str
                   /\ States (nth i (Procs M) d_lts) (nthS i s1)
                   /\ States (nth i (Procs M) d_lts) (nthS i s2)
          (* t̄[i] ≠ • ∧ s̄[i] −t̄[i]→_i s̄′[i]  *)
        | Some x => Trans (nth i (Procs M) d_lts) x (nthS i s1) (nthS i s2)
      end.

(* Transition in a system LTS, i.e. ∀s̄,s̄′ ∈ S_M : (∃(t̄,a) ∈ V : s̄ −t̄ a→ s̄′) ⟹ s̄ −a→ s̄′  *)
Definition NetTrans {n}(M : LTSNetwork)(a : Act)(s1 s2 : State n) :=
  exists v, act v = a /\ NetTransSL M v s1 s2.

(* I_M = {⟨s_1,...,s_n⟩ | ∀i ∈ 1..n : s_i ∈ I_i}  *)
Definition NetInit {n}(M : @LTSNetwork n)(s : State n) :=
  forall i, i < n -> Init (nth i (Procs M) d_lts) (nthS i s).

(* The System LTS *)
Definition SystemLTS {n}(M : LTSNetwork) :=
  Build_LTS n (NetStates M) (NetActs M) (NetTrans M) (NetInit M)
            (NetLTS_trans_cond M) (NetLTS_init_cond M)
            (NetLTS_init_not_empty_cond M) (NetLTS_tau_cond M).
```

Listing B.9: Formalization of the system LTS of an LTS network (Definition 8) in Coq

A rule system $\Sigma = (R, \hat{\mathcal{V}})$ is defined in Definition 10. In Listing B.10 the formalization of a rule system is given. Since the rule system is only used in the context of a transformation of an LTS network, the set of transformation rules is defined as a vector. In the formalization of the transformation of an LTS network we will see that each `nth i (Rules RS) d_tr` of a rule system `RS` is matched on process `nth i (Procs M) d_lts` of an LTS network `M`. The dummy transformation rule `d_tr` is a transformation rule that does not apply any changes. More specifically, the dummy transformation rule transforms patterns of the dummy LTS `d_lts` to `d_lts`.

The sets `SyncL` and `SyncR` represent the sets of synchronisation laws $\mathcal{V}$ and $\hat{\mathcal{V}}$ respectively. The `SyncLActs_cond` and `SyncRActs_cond` conditions, and `AdmL_cond` and `AdmR_cond` conditions ensure that the `SyncActs_cond` and `Adm_cond` conditions respectively hold for LTS networks generated from the `RuleSystem`. The formalization of the filtered left pattern network has the following signature: `LeftNetwork n(RS : @RuleSystem n)(I : Ensemble nat)`. The formalization of the filtered left pattern network has a similar signature.

```
(* Rule system (n-vector) *)
Record RuleSystem {n} := {
  Rules : listn TransRule n; (* Vector of transformation rules r̄
                                with ∀i ∈ 1..n : r̄[i] ∈ R  *)

  SyncL : Ensemble (@SyncLaw n); (* Set of existingsynchronisation laws V *)
  SyncR : Ensemble (@SyncLaw n); (* Set of new synchronisation laws V̂ *)

  SyncLActs_cond : NoUnknownSyncActs SyncL (map L Rules);
  SyncRActs_cond : NoUnknownSyncActs (Union SyncLaw SyncL SyncR) (map R Rules);

  AdmL_cond : Admissible SyncL (map L Rules);
  AdmR_cond : Admissible (Union SyncLaw SyncL SyncR) (map R Rules)
}.
```

Listing B.10: Formalization of a rule system (Definition 10) in Coq as a vector of transformation rules

Shown in Listing B.11 is the Coq specification for Definition 12, the transformation of an LTS network. The LTS network `TM` represents the LTS network $T_\Sigma(\mathcal{M})$ created by transforming an LTS network $\mathcal{M}$ by applying a transformation system Σ. The ith rule `nth i (Rules RS) d_tr` of a rule system `RS` is matched on process `nth i (Procs M) d_lts` of an LTS network `M` via match `mLs i`.

The SyncN_cond condition expresses that the synchronisation laws of the transformed LTS network TM is the union of the laws of the input LTS network and the laws introduced by the rule system. The SyncMatch_cond condition states that the synchronisation laws of the transformation system must match the transformation laws of the model. For now we require that the set of synchronisation laws SyncL RS and SyncN M are equivalent, where RS is a rule system and M is an LTS network. In the future, we may formalize a more flexible way to match synchronisation laws of a rule system with synchronisation laws of an LTS network.

The SyncMatch_cond condition expresses that the ith process LTSs nth i (Procs TM) d_lts is represented by the transformation of ith process TrG i ($T_i(\Pi[i])$). This transformation includes match $\hat{m}_i : \mathcal{S}_\mathcal{R} \to \mathcal{S}_{T(\Pi[i])}$, i.e. mR (TrG i).

```
(* Specification of network transformation *)
Record NetTransform {n}{M : @LTSNetwork n}{RS : @RuleSystem n} := {
  TM :> LTSNetwork; (* T_Σ(M) *)

  (* r̄[i] ∈ R has corresponding match m_i : S_L → S_Π[i] *)
  mLs (i : nat) : Match (L (nth i (Rules RS) d_tr)) (nth i (Procs M) d_lts);

  (* A RuleSystem can only be applied if the left-patterns sync laws match
     those of the network *)
  SyncMatch_cond : SyncL RS = SyncN M;
  SyncN_cond : SyncN TM = Union SyncLaw (SyncN M) (SyncR RS); (* Laws V ∪ V̂ *)

  (* T_i(Π[i]) (this includes m̂_i : S_R → S_T(Π[i])) *)
  TrG (i : nat) : Transform (mLs i);
  Procs_cond : forall i, i < n -> nth i (Procs TM) d_lts = TrG i
}.
```

Listing B.11: Formalization of the transformation of an LTS network (Definition 12) in Coq

Lemma 5 states that when a transition in a process LTS $\Pi[i]$ (with $i \in 1..n$) is matched it follows that all transitions participating in the synchronisation law are matched as well. The formalization of Lemma 5 is shown in Listing B.12.

```
Lemma Ac_Fully_Matched :
  forall {n} {M} {RS} (TrM : @NetTransform n M RS) v s s',
    (* s̄ --t̄ a-->_M s̄' *)
    NetTransSL M v s s' ->
    (* There is a matching transition in a process LTS *)
      (exists i, exists q q', mLs TrM i q (nthS i s)           (* m(q) = s̄[i] *)
              /\ mLs TrM i q' (nthS i s')                     (* m(q') = s̄'[i] *)
              /\ exists a, (nth i (sync v) None) = Some a     (* t̄[i] = a *)
                    /\ Trans (L (nth i (Rules RS) d_tr)) a q q')  (* q --a-->_L_i q' *)
            (* All participating transition are matched *)
        -> (forall i a, (nth i (sync v) None) = Some a ->
              exists q q', mLs TrM i q (nthS i s)
                         /\ mLs TrM i q' (nthS i s')
                         /\ Trans (L (nth i (Rules RS) d_tr)) a q q').
```

Listing B.12: Formalization of Lemma 5 in Coq

The formalization of Definition 15, the mapping of state vectors of pattern networks to state vectors of LTS networks, is presented in Listing B.13. The filtered pattern network $\mathcal{L}_I$ is expressed by LeftNetwork RS I, with RS a rule system and I a set of indices. A vector state p of the filtered pattern network LeftNetwork RS I is mapped to a vector state s of an LTS network M iff for all i < n /\ I i ($i \in I \cap 1..n$) nthS i p is matched on nthS i s, i.e. $m_i(\bar{p}[i]) = \bar{s}[i]$ or formalized: mLs TrM i (nthS i p) (nthS i s). We have $\bar{p} \vdash_I \bar{s}$ iff MapL TrM I p s holds. Mapping of the right filtered pattern network MapR is defined similarly. It is possible to formalize a mapping of filtered LTS networks to LTS networks as proposed in Definition 15. Such a formalization would require adding a parameter for the matching function mLs TrM and may be an option for future work.

```
(* Mapping of state vectors in S_{L_I} to state vectors in M *)
Definition MapL {n}{M : LTSNetwork}{RS : RuleSystem}(TrM : @NetTransform n M RS)
                (I : Ensemble nat)(p s : State n) :=
        (* s̄ ∈ S_M ∧ p̄ ∈ S_{L_I} *)
        States (SystemLTS M) s /\ States (SystemLTS (LeftNetwork RS I)) p
        (* ∀i ∈ I : m(p̄[i]) = s̄[i] *)
    /\ forall i, i < n -> I i -> mLs TrM i (nthS i p) (nthS i s).
```

Listing B.13: Formalization of the mapping of state vectors (Definition 15) in Coq

The formalization of Lemma 7 is given in Listing B.14. The lemma states that for each transition in a filtered pattern network there is a transition in the LTS network on which the former transition is mapped. In a way, this is similar to the third condition of a match (Definition 4). Lemma 8 is formalized similarly.

```
Lemma MapR_trans2 :
  forall {n}{M}{RS}(TrM : @NetTransform n M RS) (I : Ensemble nat) a p1 p2 s1,
    (* ∀p̄1, p̄2 ∈ S_{R_I}, s̄1 ∈ S_{T_Σ(M)} : p̄1 -a->_{R_I} p̄2 ∧ p̄1 ⊢_I s̄1 ⟹ *)
    Trans (SystemLTS (RightNetwork RS I)) a p1 p2 -> MapR TrM I p1 s1 ->
     (* ∃s̄2 ∈ S_{T_Σ(M)} : p̄2 ⊢_I s̄2 ∧ s̄1 -a->_{T_Σ(M)} s̄2 ∧ ∀i ∈ 1..n \ I : s̄1[i] = s̄2[i] *)
      exists s2, MapR TrM I p2 s2 /\ Trans (SystemLTS TrM) a s1 s2
            /\ forall i, i < n -> ~I i -> nthS i s1 = nthS i s2.
```

Listing B.14: Formalization of Lemma 7 in Coq

Given an LTS network $\mathcal{M}$ and a rule system $\Sigma = (R, \hat{\mathcal{V}})$ Lemma 9 expresses that there is a vector state $\bar{q} \in \mathcal{S}_{\mathcal{L}_I}$ such that $\bar{q} \vdash_I \bar{s}$ if there is a match with $m_i(\bar{q}[i]) = \bar{s}[i]$ for all $i \in I$, where $\bar{s} \in \mathcal{S}_{\mathcal{M}}$ and $I \subseteq 1..n$. Listing B.15 shows the formalization of Lemma 9. The last part of the conjunction states that all sub-states that are not indexed by I are dummy states. For some proofs it is helpful to know that sub-states not indexed by I are dummy states, e.g. when we have to show equivalence of sub-states of two different vector states of filtered pattern networks.

```
Lemma exists_mapLState :
  forall {n}{M}{RS}(TrM : @NetTransform n M RS) (I : Ensemble nat) s,
    (* ∀s̄ ∈ S_M ∧ (∀i ∈ I : i ∈ M(s̄)) ⟹ *)
    States (SystemLTS M) s -> (forall i, I i -> NetMatchIds TrM s i) ->
      (* ∃q̄ ∈ S_{L_I} : q̄ ⊢_I s̄ ∧ ∀i ∈ 1..n \ I : s̄[i] = * *)
      exists q, MapL TrM I q s /\ forall i, i < n -> ~I i -> Init d_lts (nthS i q).
```

Listing B.15: Formalization of Lemma 9 in Coq

The formalization of Definition 16, the κ-extended rule system, is presented in Listing B.16. Let `RS` be a rule system, `k_fun` be a function producing κ-actions from sub-states, and `k_fun2` be a function that produces unique κ-synchronisation-result actions ($\kappa_{\bar{s}}$) from a κ-synchronisation vector. The set of κ-synchronisation laws is represented by `SyncK RS k_fun k_fun2`. A synchronisation law `v` is a κ-synchronisation law iff `SyncK RS k_fun k_fun2 v` holds. The specification of the κ-extension of a rule system `RS` is formalized by `RuleSystemK RS k_fun k_fun2`. The κ-extended rule system is described by `rsk`. Together `rki` and `r_cond` state that `rki i` is the κ-extension of the ith transformation rule of `RS`. The `SyncL_cond` and `SyncR_cond` conditions specify that the κ-synchronisation laws are combined with `SyncL` ($\mathcal{V}$) and `SyncR` ($\hat{\mathcal{V}}$) of rule system `RS` to obtain the κ-extension of the sets of synchronisation laws.

```
(* The kappa synchronisation law, i.e. (⟨κ1,...,κn⟩,κ_s̄) *)
Definition SyncK {n} (RS : @RuleSystem n)(k_fun : KFun)
                 (k_fun2 : listn (option Act) n -> Act)(v : SyncLaw) :=
     (act v) = k_fun2 (sync v)   (* a = κ_s̄*)
     (* The exists a κ-synchronisation vector such that t̄ = ⟨κ1,...,κn⟩ *)
  /\ (exists I s i, i < n /\ I i
                 /\ Init (L (nth i (Rules RS) d_tr)) (nthS i s)
                 /\ sync v = KSyncVec k_fun I s).

(* System of κ-extended transformation rules *)
```

```
Record RuleSystemK {n}{RS : @RuleSystem n}{k_fun : KFun}(k_fun2 : KFun2 RS k_fun)
    := {
  rsk :> RuleSystem; (* κ-extended rule system *)

  (* The i^th κ-extended transformation rule *)
  rki (i : nat) : TransRuleK (nth i (Rules RS) d_tr) k_fun;
  r_cond  : forall i, i < n -> nth i (Rules rsk) d_tr = rki i;

  (* V^k = V ∪ SyncK, where
     SyncK = {(⟨κ1,...,κn⟩, κs̄) | s̄ ∈ I_M ∧ (∀i ∈ 1..n : κi = κ_s̄[i] ∨ κi = •)} \ {⟨•,...,•⟩} *)
  SyncL_cond : forall v, (SyncL rsk) v <-> (SyncL RS) v \/ SyncK RS k_fun k_fun2 v;
  SyncR_cond : forall v, (SyncR rsk) v <-> (SyncR RS) v \/ SyncK RS k_fun k_fun2 v
}.
```

Listing B.16: Formalization of the κ-extension of a rule system (Definition 16) in Coq

Lemma 11 expresses that initial states of filtered pattern networks are related to themselves by bisimulation relations relating the κ-extended left and right filtered pattern networks. In Listing B.17 the formalization of Lemma 11 is given. The bisimulation relation `BI_rel` is constructed from the relation between the left and right κ-extended filtered pattern networks. The `BI_rel` relation relates left and right filtered pattern networks.

```
Lemma BI_rel_glue_states_related :
  forall {n} {RS} {k_fun} {k_fun2} (RSk : @RuleSystemK n RS k_fun k_fun2) I,
    let SysL (R : RuleSystem) := SystemLTS (LeftNetwork R I) in
    let SysR (R : RuleSystem) := SystemLTS (RightNetwork R I) in
      (* forall bisimulation relations BI between L^κ_I and R^κ_I, and (forall) s̄ ∈ I_{L_I},
         s̄ is related to itself by BI_rel *)
      forall (BI : BB (SysL RSk) (SysR RSk)) s, Init (SysL RS) s ->
          (BI_rel RSk I BI) s s.
```

Listing B.17: Formalization of Lemma 11 in Coq

Lemma 12 expresses that, if a vector state $\bar{s} \in \mathcal{S}_{\mathcal{L}_I}$, of which some sub-states are glue-states, is related to $\bar{p} \in \mathcal{S}_{\mathcal{R}_I}$, then there exists a state $\hat{\bar{p}} \in \mathcal{S}_{\mathcal{R}_I}$ with the same glue-sub-states as $\bar{s}$ such that $\bar{p} \Rightarrow_{\mathcal{R}_I} \hat{\bar{p}}$ and $\hat{\bar{p}}$ is related to $\bar{s}$ as well. The formalization of Lemma 6 is presented in Listing B.18. Lemma 12 assumes $J \subseteq I \subseteq 1..n$. However, for the formalization we have chosen to use `1..n∩I∩J` for J such that $i \in J$ is represented by `i < n -> I i -> J i`.

```
Lemma TauPath_glue_state_prop :
  forall {n} {RS} {k_fun} {k_fun2} (RSk : @RuleSystemK n RS k_fun k_fun2)
         (I J : Ensemble nat),
    let SysL (R : RuleSystem) := SystemLTS (LeftNetwork R I) in
    let SysR (R : RuleSystem) := SystemLTS (RightNetwork R I) in
      forall (B : BB (SysL RSk) (SysR RSk)) s p,
          (* s̄ B_I p̄ ∧ (∀i ∈ I ∩ J : s̄[i] ∈ I_{L_i}) *)
          B s p -> (forall i, i < n -> I i -> J i ->
                       Init (L (nth i (Rules RS) d_tr)) (nthS i s))
           (* ⟹ ∃p̂ ∈ S_{R_I} : p̄ ⇒_{R_I} p̂ ∧ s̄ B_I p̂ ∧ ∀i ∈ I ∩ J : p̂[i] = s̄[i] *)
           -> exists ph, TauPath (SysR RS) p ph /\ B s ph
                        /\ (forall i, i < n -> I i -> J i ->
                              nth i ph d_str = nth i s d_str).
```

Listing B.18: Formalization of Lemma 12 in Coq

The assumption that rule systems are cascading (Definition 17) is formalized as shown in Listing B.19. Given an rule system $\Sigma = (R, \hat{\mathcal{V}})$ for all $I \subseteq 1..n$ let B_I be a relation such that $\mathcal{L}_I \leftrightarrows_b \mathcal{R}_I$. A rule system $\Sigma = (R, \hat{\mathcal{V}})$ is cascading if for all $I \subseteq 1..n$ and $J \subseteq I$ it holds that: if two states $\bar{s} \in \mathcal{S}_{\mathcal{L}_I}$ and $\bar{p} \in \mathcal{S}_{\mathcal{R}_I}$ are related by bisimulation relation B_I and both contain identical glue-states index by $I \setminus J$, then $\bar{s}' \in \mathcal{S}_{\mathcal{L}_J}$ and $\bar{p}' \in \mathcal{S}_{\mathcal{R}_J}$ are related by a bisimulation relation B_J, where $\bar{s}'$ and $\bar{p}'$ are obtained by removing the glue-states index by $I \setminus J$ from $\bar{s}$ and $\bar{p}$ respectively.

```
Axiom BBRuleSystem_cascading :
  forall {n} {RS} {k_fun} {k_fun2} (RSk : @RuleSystemK n RS k_fun k_fun2) I J
       (BI : BB (SystemLTS (LeftNetwork RSk I)) (SystemLTS (RightNetwork RSk I))),
    (* J ⊆ I *)
    Included nat J I ->
      exists (BJ : BB (SystemLTS (LeftNetwork RSk J))
                      (SystemLTS (RightNetwork RSk J))),
        (* ∀s̄ ∈ S_{L_I}, p̄ ∈ S_{R_I} : s̄ B_I p̄ ∧ (∀i ∈ I \ J : s̄[i] ∈ I_{L_i} ∧ s̄[i] = p̄[i]) *)
        forall s p, BI s p ->
          (forall i, I i -> ~J i -> Init (L (nth i (Rules RS) d_tr)) (nthS i s)
                                  /\ nthS i s = nthS i p)
          (* ⟹ ∀s̄' ∈ S_{L_J}, p̄' ∈ S_{R_J} : (∀i ∈ J : s̄[i] = s̄'[i] ∧ p̄[i] = p̄'[i]) ⟹ s̄' B_J p̄' *)
          -> forall s' p', (forall i, i < n -> J i -> nthS i s' = nthS i s
                                                    /\ nthS i p' = nthS i p
                          ) -> BJ s' p'.
```

Listing B.19: Assumption that rule systems are cascading according to Definition 17 formalized in Coq

The formalization of bisimulation relation C proposed in Proposition 2 is presented in Listing B.20. The part where matched states are related is represented by `Bhat_rel`. The part where non-matched states are related is represented by `D_rel`. Again, the bisimulation relation `BI_rel` is constructed from the relation between the left and right κ-extended filtered pattern networks. Recall $M(\bar{s})$ the set of indices of elements in $\bar{s}$ matched on by the corresponding transformation rule. The set $M(\bar{s})$ is expressed by `NetMatchIds TrM s`, i.e. $i \in M(\bar{s})$ iff `NetMatchIds TrM s i`.

```
Definition Bhat_rel {n}{M : LTSNetwork}{RS : RuleSystem}{k_fun : KFun}
                    {k_fun2 : KFun2 RS k_fun}(RSk : RuleSystemK k_fun2)
                    (TrM : @NetTransform n M RS) s p :=
    let I := NetMatchIds TrM s in
    let SysL X := SystemLTS (LeftNetwork X I) in
    let SysR X := SystemLTS (RightNetwork X I) in
      (* ∃s̄_m ∈ S_{L_{M(s̄)}}, p̄_m ∈ S_{R_{M(s̄)}} *)
      exists sm pm (B : BB (SysL RSk) (SysR RSk)),
           (* s̄_m ⊢_{M(s̄)} s̄ ∧ p̄_m ⊢_{M(s̄)} p̄ *)
           MapL TrM I sm s /\ MapR TrM I pm p
           (* s̄_m B_{M(s̄)} p̄_m *)
        /\ BI_rel RSk I B sm pm.

Definition D_rel {n}{M : LTSNetwork}{RS : RuleSystem}(TrM : @NetTransform n M RS)
                  (i : nat)(s p : State n) :=
    (* \bs[i] \in \states{i} \wedge \bs[i] = \bp[i] *)
    States (nth i (Procs M) d_lts) (nthS i s) /\ nthS i s = nthS i p.

(* C = {(s̄, p̄) | ∀i ∈ 1..n :   (i ∉ M(s̄) ⟹ s̄[i] ∈ S_i ∧ s̄[i] = p̄[i])
                               ∧(i ∈ M(s̄) ⟹ ∃s̄_m ∈ S_{L_{M(s̄)}}, p̄_m ∈ S_{R_{M(s̄)}} :
                                     s̄_m B_{M(s̄)} p̄_m ∧ s̄_m ⊢_{M(s̄)} s̄ ∧ p̄_m ⊢_{M(s̄)} p̄) } *)
Definition C_rel {n}{M : LTSNetwork}{RS : RuleSystem}{k_fun : KFun}
                  {k_fun2 : KFun2 RS k_fun}(RSk : RuleSystemK k_fun2)
                  (TrM : @NetTransform n M RS)(s p : State n) :=
  forall i, i < n ->   (~NetMatchIds TrM s i -> (D_rel TrM i) s p)
                    /\ (NetMatchIds TrM s i -> (Bhat_rel RSk TrM) s p).
```

Listing B.20: Coq formalization of generic bisimulation relation C that is defined in Proposition 2

Formalization of Proposition 2 is presented in Listing B.21. Axiom `BBNet_congruence_prop` expresses the that LTS networks are admissible with respect to τ-transitions and that branching bisimulation is a congruence for admissible LTS networks. The `BBRuleSystem_cascading` represents the assumption that rule systems are cascading. Furthermore, the `BBNetAssumption` axiom represents the pre-condition that there exists a bisimulation relation B_I between $\mathcal{L}_I^\kappa$ and $\mathcal{R}_I^\kappa$ for all $I \subseteq 1..n$. Finally, the proposition `Check_Correct` states that there exists a bisimulation relation `C` that relates an LTS network `M` and the transformed LTS network `TrM`.

```
(* An admissible LTS network is a congruence w.r.t branching bisimulation *)
Axiom BBNet_congruence_prop :  (* Code omitted *)
(* A rule system must be cascading *)
Axiom BBRuleSystem_cascading : (* Code omitted *)
(* Pre-condition: (∀I ⊆ 1..n : L_I^κ ↔_b R_I^κ)  *)
Axiom BBNetAssumption :
forall {n} {RS} {k_fun} {k_fun2} (RSk : @RuleSystemK n RS k_fun k_fun2) I,
  exists (B : BB (SystemLTS (LeftNetwork RSk I)) (SystemLTS (RightNetwork RSk I))),
       True.

(* C is a branching bisimulation relation between M (M) and TrM (T_Σ(M)) *)
Proposition Check_Correct :
  forall {n} {M} {RS} {k_fun} {k_fun2} (RSk : @RuleSystemK n RS k_fun k_fun2)
         (TrM : @NetTransform n M RS),
    exists (C : BB (SystemLTS M) (SystemLTS TrM)), True.
```

Listing B.21: Formalization of Proposition 2 in Coq