




A formal verification technique for behavioural model-to-model transformations

Sander de Putter¹ and Anton Wijs¹ 

¹Eindhoven University of Technology, Eindhoven, The Netherlands

Abstract. In Model Driven Software Engineering, models and model transformations are the primary artifacts when developing a software system. In such a workflow, model transformations are used to incrementally transform initial abstract models into concrete models containing all relevant system details. Over the years, various formal methods have been proposed and further developed to determine the functional correctness of models of concurrent systems. However, the formal verification of model transformations has so far not received as much attention. In this article, we propose a formal verification technique to determine that formalisations of such transformations in the form of rule systems are guaranteed to preserve functional properties, regardless of the models they are applied on. This work extends our earlier work in various ways. Compared to our earlier approaches, the current technique involves only up to n individual checks, with n the number of rules in the rule system, whereas previously, up to $2^n - 1$ checks were required. Furthermore, a full correctness proof for the technique is presented, based on a formal proof conducted with the Coq proof assistant. Finally, we report on two sets of conducted experiments. In the first set, we compared traditional model checking with transformation verification, and in the second set, we compared the verification technique presented in this article with the previous version.

Keywords: Model transformation verification, Explicit-state model checking, Branching bisimulation

1. Introduction

It is a well-known fact that concurrent systems are very hard to develop correctly. In order to support the development process, over the years, a whole range of formal methods have been constructed to determine the functional correctness of system models [BH04]. Over time, these techniques have greatly improved, but the analysis of complex models is still time-consuming, and often beyond what is currently possible.

To get a stronger grip on the development process, model-driven development has been proposed [KWB05]. In this approach, models are constructed iteratively, by defining *model transformations* that can be viewed as functions applicable on models: they are applied on models, producing new models. Using such transformations, an abstract initial model can be gradually transformed into a very detailed model describing all aspects of the system. If one can determine that the transformations are correct, then it is guaranteed that a correct initial model will be transformed into a correct final model.

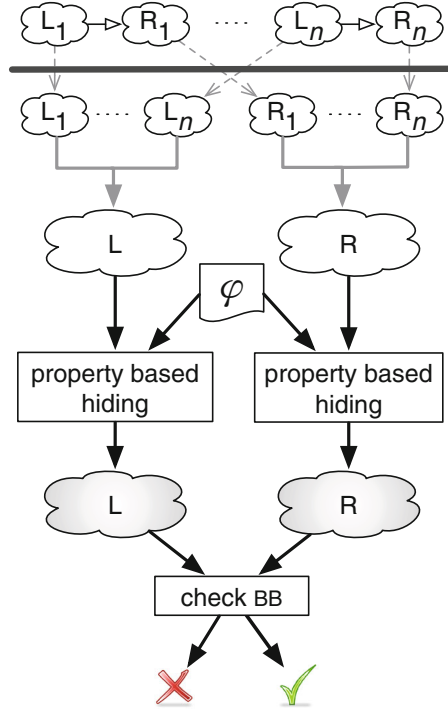


Fig. 1. LTS transformation verification with REFINER

Most model transformation verification techniques are focussed on determining that a given transformation applied on a given model produces a correct new model, but in order to show that a transformation is correct in general, one would have to determine this for *all possible input models*. There are some techniques that can do this [ACL⁺15, RW13], but it is often far from trivial to show that these are correct.

This work is an extension of [PW16], where we formally proved the correctness of such a formal transformation verification technique proposed in [WE13, Wij13] and implemented in the tool REFINER [WE14]. It is applicable on models with a semantics that can be captured by Labelled Transition Systems (LTSs). Transformations are formally defined as *LTS transformation rules*. Correctness of transformations is interpreted as the *preservation of properties*. Given a property φ written in a fragment of the μ -calculus [MW14], and a system of transformation rules Σ , REFINER checks whether Σ preserves φ for all possible inputs. This is done by first hiding all behaviour irrelevant for φ [MW14] and then checking whether the rules replace parts of the input LTSs by new parts that are *branching bisimilar* to the old ones. Branching bisimilarity preserves safety properties and a subset of liveness properties involving inevitable reachability [vGW96].

Figure 1 provides an overview of the transformation verification workflow in REFINER. Given as input is a rule system consisting of n LTS transformation rules, where each rule r_i consists of a left pattern L_i , describing which component behaviour is subject to transformation, and a right pattern R_i , defining the behaviour produced by a transformation of the corresponding left pattern behaviour. If such a rule system were to be applied on an input model, REFINER would identify the possible matches of the left patterns of the rules on the behaviour of the components in the model, and subsequently, apply transformation on those matches, thereby replacing the existing behaviour with copies of the corresponding right patterns.

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

Next, *property-based hiding* [MW14] is performed, given a property φ to check. Finally, the resulting abstract state spaces are compared using a branching bisimulation checking algorithm. Only if the combinations of both the left and the right patterns satisfy φ will the outcome of this check be positive. When no property is considered,

the technique checks for full semantics preservation, i.e., it does not apply property-based hiding. This is useful, for instance, when refactoring models.

The technique has been successfully applied to reason very efficiently about model transformations; speedups of five orders of magnitude have been measured w.r.t. traditional model checking of the models produced by a transformation [Wij13]. However, as the technique is theoretically very involved, its absolute correctness, i.e., whether it returns **true** iff a given rule system is property preserving for all possible input models, has been an open question since it was constructed. In [PW16] we first addressed the correctness of the transformation verification technique. After finding and fixing two issues the verification technique was proven correct.

Contributions. This work is an extension of [PW16] in which the formal correctness of the transformation verification technique from [Wij13] is addressed. In the current article, first of all, we have extended the expressiveness of transformation rules by distinguishing between glue-states that allow incoming and/or outgoing transitions entering or leaving the LTS patterns, respectively. By doing so, the verification technique is able to handle more cases.

Second of all, we present a new proof that shows that the required number of bisimulation checks when verifying an LTS transformation rule system can be reduced from $2^n - 1$ per set of dependent transformation rules (where n is the upper bound of the number of rules in such a sets) to only one per set of dependent rules. This proof is presented in greater detail than the one given previously [PW16]. The proof is based on a formal proof conducted with the Coq proof assistant¹ version 8.6 (December 2016). The Coq formalisation is available online.²

Structure of the article. Related work is discussed in Sect. 2. Section 3 presents the notions for and analysis of the application of a rule system consisting of only a single transformation rule. A correctness proof is presented. This section can be viewed as a first step towards discussing the complete technique, applicable on rule systems consisting of multiple rules. Next, in Sect. 4, the complete technique is presented; the discussion is continued by considering networks of concurrent process LTSs, and systems of transformation rules. Again, we give a proof of correctness.

After that, we present experimental results in Sect. 5, by which we demonstrate the effectiveness of the analysis technique, compared to, more traditional, model checking the models again once they have been altered by a model transformation. Finally, Sect. 6 contains our conclusions and pointers to future work.

2. Related work

Papers on incremental model checking (IMC) propose how to reuse model checking results of safety properties for a given input model after it has been altered [SS94, Swa96]. We also consider verifying models that are subject to changes. However, we focus on analysing transformation specifications, i.e., the changes themselves, allowing us to determine whether a change always preserves correctness, independent of the input model. Furthermore, our technique can also check the preservation of (a subset of) liveness properties.

In the context of *Dynamic graph algorithms* [EGI97], reachability is an unbounded problem [RR96, SS94], i.e., it cannot be determined solely based on the changes. Thanks to our criteria, this is not an issue in our context.

In [Sah07], an incremental algorithm is presented for updating bisimulation relations based on changes applied on a graph. Their goal is to efficiently maintain a bisimulation, whereas our goal is to assess whether bisimulations are guaranteed to remain after a transformation has been applied without considering the whole relation. As is the case for the IMC techniques, this algorithm works only for a given input graph, while we aim to prove correctness of the transformation specification itself regardless of the input.

In *refinement checking* [AL91, KLG07], supported by tools such as RODIN [ABH⁺10], FDR3³ and CSP- CASL-PROVER [KR08], it is usually checked that one model refines another. This is very similar to our approach, but refinements are defined in terms of what the new model will be, as opposed to how the new model can be obtained from the old one, i.e., model transformations are not represented as artifacts independent of the models they can be applied on. This makes the technique not directly suitable to investigate the feasibility to verify definitions of model transformations, as opposed to the models they produce.

The BART tool⁴ allows automatically refining B components to $B0$ implementations. Similar to our setting, it treats refinement rules as user-definable artifacts and performs pattern matching to do the refining. Constraints

¹ <http://coq.inria.fr>

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

³ <http://www.fsel.com/fdr3.html>.

⁴ <http://www.tools.clearsy.com/tools/bart>.

are checked to ensure that the resulting system will be correct. Other work related to B , e.g., [Lan96], is on strictly refining existing functionalities. Approaches described in, e.g., [BGL05, CCGT09, GL12, HKR⁺10] prove that a transformation preserves the semantics of any input model, by showing that the transformed model will be strong or weak bisimilar to the original. Contrary to our work, in all these approaches, no cases can be handled where transformations alter the semantics in a way that does not invalidate the functional property of interest. Furthermore, by using branching bisimilarity as opposed to strong or weak bisimilarity, our technique also supports a subset of liveness properties.

Similarly, Combemale et al. [CCGT09], Hülbusch et al. [HKR⁺10], and Karsai and Narayanan [KN07, NK08] check semantics preservation of model transformations using either strong or weak bisimilarity.

Several techniques, such as those described in [KN07, NK08, VP03], perform individual checks for each concrete model. As such, the transformation itself is not verified, but verification is done each time the transformation is applied in a concrete situation. Our technique verifies the transformation definition once, after which the verification result is relevant for each application of that transformation.

Monotonically adding functionality, as opposed to refining, is addressed in, e.g., [BE04]. The focus is on updating property formulae; it could be interesting to see if this is applicable in our setting to update properties.

In some works, e.g., [SMR11, GGL⁺06], theorem proving is used to verify the preservation of behavioural semantics. The use of theorem provers requires expert knowledge and high effort [SMR11]. In contrast, our equivalence checking approach is more lightweight, automated, and allows the construction of counter examples which help developers identify issues with the transformations.

In [BCE⁺07], transformation rules for Open Nets are verified on the preservation of dynamic semantics. Open Nets are a reactive extension of Petri Nets. The technique is comparable to our technique with two main exceptions. First, they consider weak bisimilarity for the comparison of rule patterns, which preserves a strictly smaller fragment of the μ -calculus than branching bisimilarity [MW14]. Second, their technique does not allow transforming the communication interfaces between components. Our approach allows this, and checks whether the components remain ‘compatible’.

Finally, in [SLC⁺14], transformations expressed in the DSLTrans language are checked for correspondence between source and target models. DSLTrans uses a symbolic model checker to verify properties that can be derived from the meta-models. The state space captures the evolution of the input model. In contrast, our approach considers the state spaces of combinations of transformation rules, which represent the potential behaviour described by those rules. An interesting pointer for future work is whether those two approaches can be combined.

3. Verifying single LTS transformations

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

3.1. LTS transformation and LTS equivalence

We use LTSs as in Definition 3.1 to reason about the potential behaviour of processes.

Definition 3.1 (*Labelled transition system*) An LTS \mathcal{G} is a tuple $(\mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}})$, with

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

Action labels in $\mathcal{A}_{\mathcal{G}}$ are denoted by a, b, c , etc. In addition, there is the special action label τ to represent internal, or hidden, system steps. A transition $(s, a, s') \in \mathcal{T}_{\mathcal{G}}$, or $s \xrightarrow{a}_{\mathcal{G}} s'$ for short, denotes that LTS \mathcal{G} can move from state s to state s' by performing the a -action. For the reflexive transitive closure of $\xrightarrow{a}_{\mathcal{G}}$, we use $\xrightarrow{a}_{\mathcal{G}}^*$. Note that transitions are uniquely identifiable by the combination of their source state, action label, and target state. This property is sometimes called the *extensionality* of LTSs [Win90]. This does not limit the applicability of our technique, as a system that is not extensional can be rewritten into an extensional system by introducing separate target states for each transition with an equivalent label.

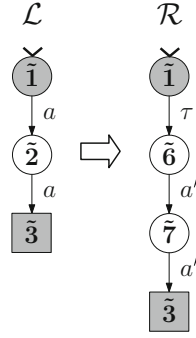


Fig. 2. A transformation rule

Finally, we only consider LTSs that are *weakly connected*, meaning that the undirected version of an LTS is a single connected component (from each state, there is path to each other state). This implies that each state in an LTS is reachable from at least one initial state. It would be irrelevant to consider states that are unreachable, since a system could never end up in such a state, starting from an initial state.

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

Definition 3.2 (Transformation rule) A transformation rule $r = (\mathcal{L}, \mathcal{R})$ consists of a left pattern LTS $\mathcal{L} = (\mathcal{S}_{\mathcal{L}}, \mathcal{A}_{\mathcal{L}}, \mathcal{I}_{\mathcal{L}}, \mathcal{E}_{\mathcal{L}})$ and a right pattern LTS $\mathcal{R} = (\mathcal{S}_{\mathcal{R}}, \mathcal{A}_{\mathcal{R}}, \mathcal{I}_{\mathcal{R}}, \mathcal{E}_{\mathcal{R}})$, with $\mathcal{I}_{\mathcal{L}} = \mathcal{I}_{\mathcal{R}}$. The two pattern LTSs are annotated with a (possibly empty) set of exit-states $\mathcal{E}_{\mathcal{L}} \subseteq \mathcal{S}_{\mathcal{L}}$ and $\mathcal{E}_{\mathcal{R}} \subseteq \mathcal{S}_{\mathcal{R}}$, respectively, with $\mathcal{E}_{\mathcal{L}} = \mathcal{E}_{\mathcal{R}}$. Finally, we must have that $\mathcal{S}_{\mathcal{L}} \cap \mathcal{S}_{\mathcal{R}} = \mathcal{I}_{\mathcal{L}} \cup \mathcal{E}_{\mathcal{L}} = \mathcal{I}_{\mathcal{R}} \cup \mathcal{E}_{\mathcal{R}}$.

The states in $\mathcal{S}_{\mathcal{L}} \cap \mathcal{S}_{\mathcal{R}}$ are called the *glue-states*. The initial (glue-)states of a pattern LTS, also called the *in-states*, represent the states at which the pattern may be entered. The *exit-states* are glue-states that represent the states from which the pattern may be left. It is possible for a glue-state to be both an in-state and an exit-state.

Figure 2 shows an example of a transformation rule $r = (\mathcal{L}, \mathcal{R})$ transforming a sequence of two a -transitions to a τ -transition followed by two a' -transitions. The initial states, i.e., the in-states of \mathcal{L} and \mathcal{R} , are indicated by an incoming arrow. The exit-states are represented by a square. Furthermore, all glue-states (i.e., the in- and exit-states) are coloured grey.

These LTS patterns expect only ingoing transitions at state $\langle \tilde{1} \rangle$ as this is an in-state. At exit-state $\langle \tilde{3} \rangle$, only outgoing transitions are expected. Our previous formalisation [PW16] cannot express these subtleties as it did not distinguish between in-states and out-states. In Sect. 3.2, we show that, due to the in-states and out-states, the formalisation presented in this article is able to tell that, when the a' -transitions are relabelled to a -transitions, this transformation rule is correct while the previous formalisation cannot.

When applying a transformation rule to an LTS, the changes are applied relative to the glue-states. To reason about the application of a transformation rule, we first define the notion of an *LTS morphism*.

Definition 3.3 (LTS morphism) An *LTS morphism* $f : \mathcal{G}_0 \rightarrow \mathcal{G}_1$ between two LTSs $\mathcal{G}_0 = (\mathcal{S}_{\mathcal{G}_0}, \mathcal{A}_{\mathcal{G}_0}, \mathcal{T}_{\mathcal{G}_0}, \mathcal{I}_{\mathcal{G}_0})$ and $\mathcal{G}_1 = (\mathcal{S}_{\mathcal{G}_1}, \mathcal{A}_{\mathcal{G}_1}, \mathcal{T}_{\mathcal{G}_1}, \mathcal{I}_{\mathcal{G}_1})$ is a pair of functions $f = (f_S : \mathcal{S}_{\mathcal{G}_0} \rightarrow \mathcal{S}_{\mathcal{G}_1}, f_T : \mathcal{T}_{\mathcal{G}_0} \rightarrow \mathcal{T}_{\mathcal{G}_1})$ which preserve source states, target states, and transition labels, i.e., for all $s \xrightarrow{a}_{\mathcal{G}_0} s'$, it holds that $f_T(s \xrightarrow{a}_{\mathcal{G}_0} s') = f_S(s) \xrightarrow{a}_{\mathcal{G}_1} f_S(s')$.

It should be noted that for extensional LTSs, there is never a need to explicitly indicate how transitions are mapped by an LTS morphism f . In extensional LTSs, no two transitions have the same source state, label, and target state. This ensures that given a function $f_S : \mathcal{S}_{\mathcal{G}_0} \rightarrow \mathcal{S}_{\mathcal{G}_1}$, an LTS morphism f is implied by it, since no two transitions in \mathcal{G}_0 can be mapped to the same transition in \mathcal{G}_1 , i.e., a function $f_T : \mathcal{T}_{\mathcal{G}_0} \rightarrow \mathcal{T}_{\mathcal{G}_1}$ is implied. Because of that, with slight abuse of notation, we directly reason about LTS morphisms f as mappings between LTS states, in the remainder of this article.

A transformation rule $r = (\mathcal{L}, \mathcal{R})$ is *applicable* on an LTS \mathcal{G} iff a *match* $m : \mathcal{L} \rightarrow \mathcal{G}$ exists according to Definition 3.4. Given a state $s \in \mathcal{S}_{\mathcal{G}}$ of an input LTS and a state $p \in \mathcal{S}_{\mathcal{P}}$, we write $m(p) = s$ to indicate that state s is matched on by state p via match m . The set $m(S) = \{m(s) \in \mathcal{S}_{\mathcal{G}} \mid s \in S\}$ is the image of a set of states $S \subseteq \mathcal{S}_{\mathcal{P}}$ through match m on an LTS \mathcal{G} .

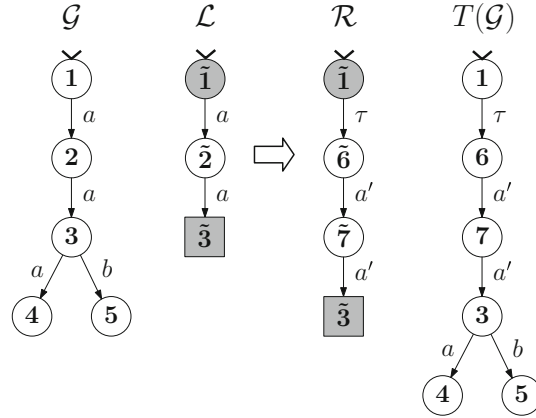


Fig. 3. Application of a transformation rule

Definition 3.4 (Match) A pattern LTS $\mathcal{P} = (\mathcal{S}_{\mathcal{P}}, \mathcal{A}_{\mathcal{P}}, \mathcal{T}_{\mathcal{P}}, \mathcal{I}_{\mathcal{P}})$ with a set of exit-states $\mathcal{E}_{\mathcal{P}}$ has a *match* $m : \mathcal{P} \rightarrow \mathcal{G}$ on an LTS $\mathcal{G} = (\mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}})$ iff m is an injective LTS morphism and for all $p \in \mathcal{S}_{\mathcal{P}}, s \in \mathcal{S}_{\mathcal{G}}$:

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

A match is a behaviour preserving morphism of a pattern LTS \mathcal{P} in an LTS \mathcal{G} defined via a category of LTSs [Win90]. The first match condition expresses that an initial state may only be matched on by exit-states. This is a reasonable assumption as all reachable behaviour starts at initial states. A consequence of the condition is that initial states may not be removed by a transformation.

The remaining two conditions make sure that a match may not cause removal of transitions that are not explicitly present in \mathcal{P} . The first condition ensures that if a match of the pattern LTS is entered at some state $s \in \mathcal{S}_{\mathcal{G}}$, then the state matching s must be an in-state. Similarly, the second condition states that if a match of the pattern LTS is left at some state $s \in \mathcal{S}_{\mathcal{G}}$, then the state matching s must be an exit-state.

An LTS \mathcal{G} is transformed to an LTS $T(\mathcal{G})$ according to Definition 3.5. For clarity, we refer with p, p', \dots to states in a left pattern LTS, with q, q', \dots to states in a right pattern LTS, with s, s', \dots to states in an input LTS, and with t, t', \dots to states in an output LTS.

Definition 3.5 (LTS transformation) Let $\mathcal{G} = (\mathcal{S}_{\mathcal{G}}, \mathcal{A}_{\mathcal{G}}, \mathcal{T}_{\mathcal{G}}, \mathcal{I}_{\mathcal{G}})$ be an LTS and let $r = (\mathcal{L}, \mathcal{R})$ be a transformation rule with match $m : \mathcal{L} \rightarrow \mathcal{G}$. Moreover, consider match $\hat{m} : \mathcal{R} \rightarrow T(\mathcal{G})$, with $\forall q \in \mathcal{S}_{\mathcal{L}} \cap \mathcal{S}_{\mathcal{R}}. \hat{m}(q) = m(q)$ and $\forall q \in \mathcal{S}_{\mathcal{R}} \setminus \mathcal{S}_{\mathcal{L}}. \hat{m}(q) \notin \mathcal{S}_{\mathcal{G}}$, by which \hat{m} defines the new states being introduced by the transformation. The *transformation* of LTS \mathcal{G} , via rule r with matches m, \hat{m} , is defined as $T(\mathcal{G}) = (\mathcal{S}_{T(\mathcal{G})}, \mathcal{A}_{T(\mathcal{G})}, \mathcal{T}_{T(\mathcal{G})}, \mathcal{I}_{\mathcal{G}})$ where

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

Given a match, an LTS transformation replaces states and transitions matched by \mathcal{L} by a copy of \mathcal{R} yielding LTS $T(\mathcal{G})$. An application of a transformation rule is shown in Fig. 3. Again, the initial states are indicated by an incoming arrow. In the middle of Fig. 3, the transformation rule $r = (\mathcal{L}, \mathcal{R})$ is shown (presented earlier in Fig. 2) which is applied on LTS \mathcal{G} resulting in LTS $T(\mathcal{G})$. The states are numbered such that matches can be identified by the state label, i.e., a state \tilde{i} is matched onto state i . Note that such a match satisfies the conditions of Definition 3.4: State $\langle \tilde{1} \rangle$ is not an exit-state, but state $\langle 1 \rangle$ does not have unmatched outgoing transitions, state $\langle \tilde{3} \rangle$ is not an in-state, but there are no unmatched incoming transitions to state $\langle 3 \rangle$, and finally, state $\langle 3 \rangle$ has unmatched outgoing transitions, but this is allowed, since $\langle \tilde{3} \rangle$ is an exit-state.

On the other hand, states $\langle \tilde{1} \rangle, \langle \tilde{2} \rangle$ and $\langle \tilde{3} \rangle$ of \mathcal{L} do not match on states $\langle 2 \rangle, \langle 3 \rangle$, and $\langle 4 \rangle$, respectively, as this violates condition 2 of Definition 3.4. Namely, transition $\langle 3 \rangle \xrightarrow{b}_{\mathcal{G}} \langle 5 \rangle$ is unmatched, since state $\langle 5 \rangle$ is unmatched, but state $\langle 2 \rangle$ is not an exit-state.

Since in general, \mathcal{L} may have several matches on \mathcal{G} , we assume that transformations are *confluent*, i.e., that they are guaranteed to terminate and lead to a unique $T(\mathcal{G})$. Confluence of LTS transformations can be checked efficiently [Wij15]. By assuming confluence, we can focus on having a single match when verifying transformation rules, since the transformations of individual matches do not influence each other.

To compare LTSs, we use the *branching bisimulation* equivalence relation [vGW96] as presented in Definition 3.6. Branching bisimulation supports abstraction from actions and is sensitive to internal actions and the branching structure of an LTS. We require abstraction from actions for the verification of abstraction and refinement transformations such that input and output models can be compared on the same abstraction level.

Definition 3.6 (*Branching bisimulation*) A binary relation B between two LTSs \mathcal{G}_1 and \mathcal{G}_2 is a *branching bisimulation* iff $s B t$ implies

1. $s \xrightarrow{a}_{\mathcal{G}_1} s' \Rightarrow (a = \tau \wedge s' B t) \vee (t \xrightarrow{\tau}_{\mathcal{G}_2} \hat{t} \xrightarrow{a}_{\mathcal{G}_2} t' \wedge s B \hat{t} \wedge s' B t')$,
2. $t \xrightarrow{a}_{\mathcal{G}_2} t' \Rightarrow (a = \tau \wedge s B t') \vee (s \xrightarrow{\tau}_{\mathcal{G}_1} \hat{s} \xrightarrow{a}_{\mathcal{G}_1} s' \wedge \hat{s} B t \wedge s' B t')$

Two states $s, t \in S$ are *branching bisimilar*, denoted $s \leftrightarrow_b t$, iff there is a branching bisimulation B such that $s B t$. Two sets of states S_1 and S_2 are called *branching bisimilar*, denoted $S_1 \leftrightarrow_b S_2$, iff $\forall s_1 \in S_1. \exists s_2 \in S_2. s_1 \leftrightarrow_b s_2$ and vice versa. We say that two LTSs \mathcal{G}_1 and \mathcal{G}_2 are *branching bisimilar*, denoted $\mathcal{G}_1 \leftrightarrow_b \mathcal{G}_2$, iff $\mathcal{I}_{\mathcal{G}_1} \leftrightarrow_b \mathcal{I}_{\mathcal{G}_2}$.

3.2. Analysing a transformation rule

The basis of the transformation verification procedure is to check whether the two patterns making up a transformation rule are equivalent, while respecting that these patterns represent embeddings in larger systems. We want to be able to verify the transformation's side effects on both the matched states and the states connected to these matched states. To make this explicit, we extend the left- and right-patterns of a transformation rule $r = (\mathcal{L}, \mathcal{R})$ according to Definition 3.7. The resulting so-called κ -extended transformation rule is defined as $r^\kappa = (\mathcal{L}^\kappa, \mathcal{R}^\kappa)$, and is specifically used for the purpose of analysing r , it does not replace r .

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

Definition 3.7 (κ -extension of a pattern LTS) The pattern LTS \mathcal{P} extended with a κ -state, and σ -, ε - and γ -transitions is defined as:

$$\begin{aligned} \mathcal{P}^\kappa &= (\mathcal{S}_{\mathcal{P}} \cup \{\kappa\}, \mathcal{A}_{\mathcal{P}} \cup \{\sigma_p \mid p \in \mathcal{I}_{\mathcal{P}}\} \cup \{\varepsilon_{p'} \mid p' \in \mathcal{E}_{\mathcal{P}}\} \cup \{\gamma_{p,p'} \mid p \in \mathcal{E}_{\mathcal{P}} \wedge p' \in \mathcal{I}_{\mathcal{P}}\}, \\ &\quad \mathcal{I}_{\mathcal{P}} \cup \{\kappa \xrightarrow{\sigma_p} p \mid p \in \mathcal{I}_{\mathcal{P}}\} \cup \{p \xrightarrow{\varepsilon_{p'}} \kappa \mid p \in \mathcal{E}_{\mathcal{P}}\} \cup \{p \xrightarrow{\gamma_{p,p'}} p' \mid p \in \mathcal{E}_{\mathcal{P}} \wedge p' \in \mathcal{I}_{\mathcal{P}}\}, \{\kappa\} \cup \mathcal{E}_{\mathcal{P}}) \end{aligned}$$

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

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

If the original \mathcal{L} and \mathcal{R} are branching bisimilar, then one cannot in general conclude that input and output LTSs on which the rule $r = (\mathcal{L}, \mathcal{R})$ is applicable are branching bisimilar as well. For instance, consider the transformation rule in Fig. 4 which swaps a and b transitions. Without the κ -extensions, the LTS patterns are branching bisimilar. However, this would not capture the fact that patterns should be interpreted as possible embeddings in larger LTSs. These larger LTSs may not be branching bisimilar, because glue-states $\langle \hat{2} \rangle$ and $\langle \hat{3} \rangle$ could be mapped to states with different outgoing transitions, apart from the behaviour described in the LTS patterns (states $\langle \hat{2} \rangle$ and $\langle \hat{3} \rangle$ are exit-states). However, due to the introduced κ -state and in particular the ε -transitions, a comparison of the κ -extended networks is able to determine that the rule does not guarantee branching bisimilarity between input and output LTSs.

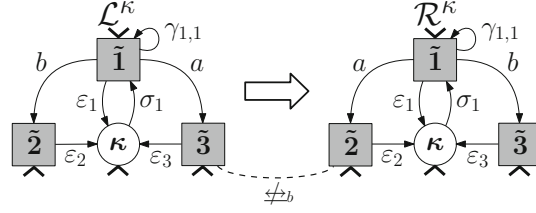


Fig. 4. The ε_3 -transitions ensure $(3) \not\sim_b (2)$

Figure 5 shows that the verification approach discussed in this article is able to perform a more fine grained analysis compared to the approach in previous work [PW16]. The κ -extension of the transformation rule in Fig. 2, but now with a' replaced by a , is shown in Fig. 5a, b using the approach presented in this article and the approach in previous work, respectively. In the latter case, the notions of in-state and exit-state are not used, instead both types of states are treated in the same way, as glue-states.

The approach discussed in this article determines that the left and right κ -extended pattern LTSs are branching bisimilar as shown in Fig. 5b. The branching bisimulation relation between the left and right κ -extended pattern LTSs is indicated with dashed lines. The introduction of the τ -transition does not break branching bisimilarity since no behaviour is lost.

However, the approach in [PW16] reports a counter-example as shown in Fig. 5b. Since the approach in [PW16] does not distinguish between in-states and exit-states, the semantics of the transformation rule is slightly different; each glue-state is allowed to be matched on states with ingoing transitions, outgoing transitions, and both in- and outgoing transitions. Therefore, any correct verification technique would have to consider the possibility that the glue-states are matched on states that have additional in- and/or outgoing transitions, and therefore, the extra τ -transition in \mathcal{R}^κ could mean that unmatched outgoing transitions are disabled when the τ -transition is followed. By adding the notions of in-state and exit-state, we can restrict the applicability of transformation rules and thereby provide more information to the verification technique.

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

1. Construct the κ -extended pattern LTSs \mathcal{L}^κ and \mathcal{R}^κ according to Definition 3.7.
2. Determine whether \mathcal{L}^κ and \mathcal{R}^κ are branching bisimilar.

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

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

In the *first* step of the verification of a rule r , a κ -state is added and for each glue state one σ -, ε , and/or γ -transition is added. The number of states added is constant, the number of transitions added is $O(g)$, and the number of action labels added is $O(g)$. Hence, the running time of step 1 is $O(g)$.

In the *second* step, it is checked whether \mathcal{L}^κ and \mathcal{R}^κ are branching bisimilar. Branching bisimilarity checking can be performed in $O(t \cdot \log(s + a))$ [GW16]. Therefore, the time complexity of the final step of the analysis is $O((t + g) \cdot \log((s + 1) + (a + g)))$.

3.3. Correctness of the verification

In this section we prove the correctness of the analysis algorithm presented in the previous section. First, we introduce two lemmas that express properties of left and right κ -extended pattern LTSs that are branching bisimilar. Next, we prove the soundness of the approach in Proposition 3.1. Finally, the completeness of the approach is proven in Proposition 3.2.

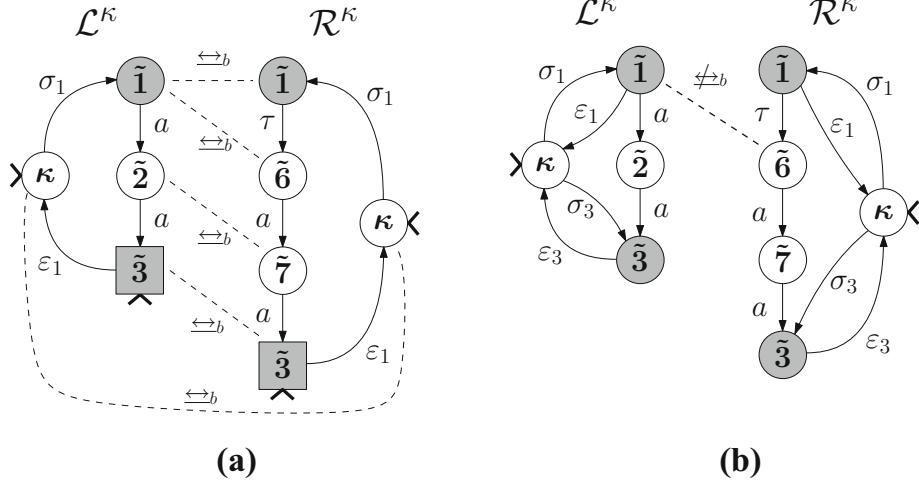


Fig. 5. The approach presented in this article (Fig. 5a) is able to determine from that the transformation rule shown in Fig. 2 (where a' has been relabelled to a) guarantees that the input and output LTSs are branching bisimilar; this is an improvement over our previous formalisation [PW16] (Fig. 5b) which reports a counter-example shown as it does not distinguish between in- and exit-states. **a** The κ -extension of the transformation rule shown in Fig. 2, relabelled with $a' := a$, using the formalisation in this article; the left and right κ -extended pattern LTSs are branching bisimilar. **b** The κ -extension of the transformation rule shown in Fig. 2, relabelled with $a' := a$, where in-states and exit-states are not distinguished from each other; the left and right κ -extended pattern LTSs are *not* branching bisimilar since in \mathcal{R}^κ , the possibility of performing a ε_1 -transition is lost once the τ -transition from state $\langle \tilde{1} \rangle$ to state $\langle \tilde{6} \rangle$ is taken

Recall that glue-states are not removed by transformation and that the κ -state represents unmatched states, and therefore also represents states that are not removed. When comparing LTS patterns by checking for branching bisimilarity, it is desirable that these states are related to themselves, as illustrated in the previous example. Lemma 3.1 shows that it is indeed the case that κ -extension achieves this: if two κ -extended pattern LTSs \mathcal{L}^κ , \mathcal{R}^κ are branching bisimilar, then the κ -state, the in-states, and the exit-states, i.e., the initial states of the κ -extended LTS patterns, are related to themselves.

Lemma 3.1 Consider a transformation rule $r = (\mathcal{L}, \mathcal{R})$ such that $\mathcal{L}^\kappa \leftrightarrow_b \mathcal{R}^\kappa$. Then, $\forall p \in \{\kappa\} \cup \mathcal{I}_{\mathcal{L}} \cup \mathcal{E}_{\mathcal{L}}, p \leftrightarrow_b p$.

Proof The proof follows from the fact that the σ - and ε -transitions are uniquely constructed for a specific glue-state. Consider a state $p \in \{\kappa\} \cup \mathcal{I}_{\mathcal{L}} \cup \mathcal{E}_{\mathcal{L}}$. By Definition 3.7, we have $p \in \mathcal{I}_{\mathcal{L}^\kappa}$. Since \mathcal{L}^κ and \mathcal{R}^κ are branching bisimilar, there is a state $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $p \leftrightarrow_b q$. We perform a case distinction on $p \in \{\kappa\} \cup \mathcal{I}_{\mathcal{L}} \cup \mathcal{E}_{\mathcal{L}}$. In each case we show that there is a transition labelled with a σ or ε between p and $p' \in \mathcal{S}_{\mathcal{L}^\kappa}$ such that the action label uniquely identifies the states p and p' . For convenience, let us refer to this unique label as α and say we have a transition $p \xrightarrow{\alpha}_{\mathcal{L}^\kappa} p'$. As $p \leftrightarrow_b q$ we can apply Definition 3.6 to show that q simulates p . As the unique labels are not allowed to be the silent action τ , the only remaining case indicates that there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\alpha}_{\mathcal{R}^\kappa} q'$ with $p \leftrightarrow_b \hat{q}$ and $p' \leftrightarrow_b q'$. There is only one transition labeled α in both \mathcal{L}^κ and \mathcal{R}^κ and it occurs as $p \xrightarrow{\alpha}_{\mathcal{L}^\kappa} p'$. It follows that $\hat{q} = p$ and $q' = p'$. Consequently, we have $p \leftrightarrow_b p$ and $p' \leftrightarrow_b p'$.

We now discuss the case distinction in full detail:

- $p = \kappa$. By Definition 3.1, $\mathcal{I}_{\mathcal{L}} \neq \emptyset$, so there is a state $p' \in \mathcal{I}_{\mathcal{L}}$. This means that there is a transition $\kappa \xrightarrow{\sigma_{p'}}_{\mathcal{L}^\kappa} p'$ where $\sigma_{p'} \neq \tau$ and $\sigma_{p'}$ uniquely occurs on $\kappa \xrightarrow{\sigma_{p'}} p'$ in both \mathcal{L}^κ and \mathcal{R}^κ (Definition 3.7). Hence, since $\sigma_{p'} \neq \tau$, by Definition 3.6, there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $\kappa \xrightarrow{\tau}_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\sigma_{q'}}_{\mathcal{R}^\kappa} q'$ with $\kappa \leftrightarrow_b \hat{q}$. The $\sigma_{p'}$ -transition in both \mathcal{L}^κ and \mathcal{R}^κ is strictly present as $\kappa \xrightarrow{\sigma_{p'}} p'$. It follows that $\hat{q} = \kappa$, therefore we have $p \leftrightarrow_b p$.
- $p \in \mathcal{I}_{\mathcal{L}}$. Then there is a transition $\kappa \xrightarrow{\sigma_p}_{\mathcal{L}^\kappa} p$ where $\sigma_p \neq \tau$ and σ_p uniquely occurs from κ to p in both \mathcal{L}^κ and \mathcal{R}^κ (Definition 3.7). In the previous case we established that $\kappa \leftrightarrow_b \kappa$. Because $\sigma_p \neq \tau$, by Definition 3.6, we have states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $\kappa \xrightarrow{\tau}_{\mathcal{R}^\kappa} \hat{q} \xrightarrow{\sigma_{q'}}_{\mathcal{R}^\kappa} q'$ with $p \leftrightarrow_b q'$. The σ_p -transition in \mathcal{L}^κ and \mathcal{R}^κ only goes from κ to p . It follows that $q' = p$ and thus $p \leftrightarrow_b p$.

- $p \in \mathcal{E}_{\mathcal{L}}$. By Definition 3.7, there is a state $p \in \mathcal{E}_{\mathcal{L}^\kappa}$ with an observable action ε_p that uniquely occurs on a transition from p to κ in both \mathcal{L}^κ and \mathcal{R}^κ . Moreover, since $\mathcal{L}^\kappa \leftrightarrow_b \mathcal{R}^\kappa$, there is a state $q \in \mathcal{I}_{\mathcal{R}^\kappa}$ such that $p \leftrightarrow_b q$. Therefore, by $\varepsilon_p \neq \tau$ and Definition 3.6, there are states $\hat{q} \in \mathcal{S}_{\mathcal{R}^\kappa}$ and $q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}_{\mathcal{R}^\kappa}^* \hat{q} \xrightarrow{\varepsilon_p}_{\mathcal{R}^\kappa} q'$ with $p \leftrightarrow_b \hat{q}$ and $\kappa \leftrightarrow_b q'$. By Definition 3.7, there is only one transition labeled ε_p in \mathcal{L}^κ and \mathcal{R}^κ , which goes from p to κ . It follows that $\hat{q} = p$ and hence $p \leftrightarrow_b p$. \square

Exit-states are the states where the embedding of an LTS pattern may be left. A transition leaving the embedding is represented in the κ -extended pattern by a ε -transition. Should an arbitrary state $q \in \mathcal{S}_{\mathcal{R}}$ be related to an exit-state $p \in \mathcal{E}_{\mathcal{L}}$, then there must exist a τ -path from q to p , otherwise state q cannot simulate the ε -transitions from p . Lemma 3.2 shows that, indeed, such a τ -path from q to p exists.

Lemma 3.2 Consider a transformation rule $(\mathcal{L}, \mathcal{R})$ such that $\mathcal{L}^\kappa \leftrightarrow_b \mathcal{R}^\kappa$, then

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

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

Let state $p \in \mathcal{E}_{\mathcal{L}}$ and state $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \leftrightarrow_b p$. By Definition 3.7 we have $p \xrightarrow{\varepsilon_p}_{\mathcal{L}} \kappa$ with $\varepsilon_p \neq \tau$. Since $p \leftrightarrow_b q$ and $\varepsilon_p \neq \tau$ there are $\hat{q}, q' \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $q \xrightarrow{\tau}_{\mathcal{R}^\kappa}^* \hat{q} \xrightarrow{\varepsilon_p}_{\mathcal{R}^\kappa} q'$ with $p \leftrightarrow_b \hat{q}$ and $\kappa \leftrightarrow_b q'$. The ε_p action only occurs on $p \xrightarrow{\varepsilon_p} \kappa$ in both \mathcal{L}^κ and \mathcal{R}^κ , therefore, we must have $\hat{q} = p$ and $q' = \kappa$. It follows that $q \xrightarrow{\tau}_{\mathcal{R}^\kappa}^* p$ and by structural induction $q \xrightarrow{\tau}_{\mathcal{R}^\kappa}^* p$. \square

Definition 3.8 introduces a mapping that formally defines how a κ -extended LTS pattern is related to the LTS that is matched on. The fact that κ -states represent all states that are not matched on is made explicit by this mapping.

Definition 3.8 (*Mapping of κ -extended LTS*) Consider an LTS \mathcal{G} and a pattern LTS \mathcal{P} with corresponding match $m : \mathcal{P} \rightarrow \mathcal{G}$. We say that a $p \in \mathcal{S}_{\mathcal{P}^\kappa}$ is *mapped* to a state $s \in \mathcal{S}_{\mathcal{G}}$, denoted by $m^\kappa(p) = s$, iff either $p \neq \kappa$ and $m(p) = s$ or $p = \kappa$ and there is no state in $\mathcal{S}_{\mathcal{L}}$ matching on s (i.e., $\neg \exists x \in \mathcal{S}_{\mathcal{P}}, m(x) = s$).

Soundness of the analysis. A transformation rule r preserves the branching structure of all LTSs it is applicable on if the κ -extended patterns of r are branching bisimilar. This is expressed in Proposition 3.1.

Proposition 3.1 is a special case of Proposition 4.1, discussed in the next section, which considers transformation of concurrent systems. This proof is derived from the Coq proof of Proposition 4.1 to explain the transformation verification technique in a more simple and intuitive setting. Lemmas 3.1 and 3.2 have been formalised in Coq.

Proposition 3.1 Let \mathcal{G} be an LTS, let r be a transformation rule with matches $m : \mathcal{L} \rightarrow \mathcal{G}$ and $\hat{m} : \mathcal{R} \rightarrow T(\mathcal{G})$ such that Definition 3.5 is satisfied. Then,

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

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

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

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

States that are touched by the transformation are related via the corresponding matches m and \hat{m} , and branching bisimulation relation B . States that are left untouched by the transformation are represented by the κ -state for which it holds that $\kappa B \kappa$ (Lemma 4.4). The mappings m^κ and \hat{m}^κ map the κ -state on all states that are not matched on. To ensure that untouched states are related to themselves we require $s = t$ whenever either s or t is mapped on by a κ -state.

We now prove that C is a branching bisimulation relation by showing that the initial states of \mathcal{G} and $T(\mathcal{G})$ are related, and that Definition 3.6 holds for C . For the latter we only discuss one of the two symmetric cases.

- C relates the initial states of \mathcal{G} and $T(\mathcal{G})$. Since we have $\mathcal{I}_{\mathcal{G}} = \mathcal{I}_{T(\mathcal{G})}$, we only have to show $\forall s \in \mathcal{I}_{\mathcal{G}}. \exists t \in \mathcal{I}_{T(\mathcal{G})}. s C t$. Take s again for t , we have to show that $s C s$. State s is either matched on or not matched on:
 - s is matched on by m , i.e., $\exists p \in \mathcal{S}_{\mathcal{L}}. m(p) = s$. We have $p \in \mathcal{E}_{\mathcal{L}}$ since initial states may only be matched on by exit-states (first condition of Definition 3.4). By Lemma 3.1 it follows that $p B p$. Hence, we have $s C s$.
 - s is not matched on by m , i.e., $\neg \exists p \in \mathcal{S}_{\mathcal{L}}. m(p) = s$. By definition we have $m^\kappa(\kappa) = s$. By Lemma 3.1 it follows that $\kappa B \kappa$. Therefore, we have $s C s$.

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

- If $s C t$ and $s \xrightarrow{a}_{\mathcal{G}} s'$ then either $a = \tau \wedge s' C t$, or $t \xrightarrow{\tau}_{T(\mathcal{G})}^* \hat{t} \xrightarrow{a}_{T(\mathcal{G})} t' \wedge s C \hat{t} \wedge s' C t'$. By definition of $s C t$, there are states $p \in \mathcal{S}_{\mathcal{L}^\kappa}$ and $q \in \mathcal{S}_{\mathcal{R}^\kappa}$ such that $p B q$, $m^\kappa(p) = s$, $\hat{m}^\kappa(q) = t$, and $(p = \kappa \vee q = \kappa) \Rightarrow s = t$ (1). Furthermore, by definition of m^κ , there is a state $p' \in \mathcal{S}_{\mathcal{L}^\kappa}$ such that $m^\kappa(p') = s'$. The transition $s \xrightarrow{a}_{\mathcal{G}} s'$ is either matched on by a transition $p \xrightarrow{a}_{\mathcal{L}} p'$ or not match on:
 1. There exists a transition $p \xrightarrow{a}_{\mathcal{L}} p'$ matching on $s \xrightarrow{a}_{\mathcal{G}} s'$ in \mathcal{L} . Since $p B q$, by Definition 3.6, we have the following two cases:
 - $a = \tau$ with $p' B q$. Since $m^\kappa(p') = s'$ and $\hat{m}^\kappa(q) = t$, we have $s' C t$.
 - $q \xrightarrow{\tau}_{\mathcal{R}} \hat{q} \xrightarrow{a}_{\mathcal{R}} q'$ with $p B \hat{q}$ and $p' B q'$. Transitions from and to κ -states are not matched on by m and \hat{m} , i.e., only transitions in $\mathcal{T}_{\mathcal{L}}(\mathcal{T}_{\mathcal{R}})$ match on transitions in $\mathcal{T}_{\mathcal{G}}(\mathcal{T}_{T(\mathcal{G})})$. Hence, states p, p', q, \hat{q} and q' cannot be κ -states. It follows that $s C \hat{m}^\kappa(\hat{q})$ and $s' C \hat{m}^\kappa(q')$, since the matching states are not κ , $m^\kappa(p) = s$, and $m^\kappa(p') = s'$. Finally, as $\hat{m}^\kappa(q) = t$, we have $t \xrightarrow{\tau}_{T(\mathcal{G})}^* \hat{m}^\kappa(\hat{q}) \xrightarrow{a}_{T(\mathcal{G})} \hat{m}^\kappa(q')$.
 2. There is no transition matching $s \xrightarrow{a}_{\mathcal{G}} s'$ in \mathcal{L} , i.e., $\neg p \xrightarrow{a}_{\mathcal{L}} p'$. Thus, both s and s' are not removed by the transformation. We distinguish two cases:
 - State s is not matched on by m . Therefore, we must have $m^\kappa(p) = s$ with $p = \kappa$. It now follows from (1) that $s = t$. Hence, $t \xrightarrow{a}_{T(\mathcal{G})} s'$, and by reflexivity of $\xrightarrow{\tau}^*$, $t \xrightarrow{\tau}_{T(\mathcal{G})}^* t \xrightarrow{a}_{T(\mathcal{G})} s'$. We have $s C t$, thus, what remains to be shown is $s' C s'$. Since s is not matched on, it follows from Definition 3.4 that state s' is either not matched on or matched on by an in-state. In the former case we have $p' = \kappa$, and in the latter case we have $p' \in \mathcal{I}_{\mathcal{L}}$. In both cases we can apply Lemma 3.1 to obtain $p' B p'$. It follows that $s' C s'$.
 - State s is matched on by a state p , i.e., $m(p) = s$. We must have $p \neq \kappa$. Since there is no transition matching $s \xrightarrow{a}_{\mathcal{G}} s'$, it follows from the second matching condition (Definition 3.4) that $p \in \mathcal{E}_{\mathcal{L}}$. Now it follows from $p B q$ and Lemma 3.2 that $q \xrightarrow{\tau}_{\mathcal{R}}^* p$. Moreover, since \hat{m} is an embedding the transition is preserved in $T(\mathcal{G})$ and we have $t \xrightarrow{\tau}_{T(\mathcal{G})}^* s \xrightarrow{a}_{T(\mathcal{G})} s'$. What is left to show is that $s C s$ and $s' C s'$. As $p \in \mathcal{E}_{\mathcal{L}}$ and there is no transition in \mathcal{L} matching $s \xrightarrow{a}_{\mathcal{G}} s'$ the state q' must be either a κ -state or an in-state, i.e., $p' \in \mathcal{I}_{\mathcal{L}} \cup \{\kappa\}$. For $p \in \mathcal{E}_{\mathcal{L}}$ and $p' \in \mathcal{I}_{\mathcal{L}} \cup \{\kappa\}$ it follows from Lemma 3.1 that $p B p$ and $p' B p'$, respectively. Hence, we have $s C t$ and $s' C s'$.
- If $s C t$ and $t \xrightarrow{a}_{T(\mathcal{G})} t'$ then either $a = \tau \wedge s C t'$, or $s \xrightarrow{\tau}_{\mathcal{G}}^* \hat{s} \xrightarrow{a}_{\mathcal{G}} s' \wedge \hat{s} C t \wedge s' C t'$. This case is symmetric to the previous case. \square

Completeness of the analysis. Completeness is an important factor in verification. A complete analysis technique will not report false negatives. The next proposition expresses that our analysis technique is complete. In the context of this work, completeness means that the analysis will always report that the left and right κ -extended pattern LTSs of a transformation rule r are branching bisimilar if the input LTS \mathcal{G} and output LTS $T(\mathcal{G})$ produced by applying r on \mathcal{G} are branching bisimilar for any given input LTS \mathcal{G} and any given matching. The proof for Proposition 3.2 is derived from the Coq proof of Proposition 4.2 to explain the transformation verification technique in a more simple and intuitive setting.

We want to stress that the analysis considers all possible input LTSs. Because of this, it may be that the analysis reports that a transformation rule does not preserve a given property in general, while the property may

still hold after transformation of some specific input LTS. Consider, for instance, a transformation rule r that is *not* property preserving according to the analysis. There may still be an input LTS \mathcal{G}_1 with a match m_1 such that $\mathcal{G}_1 \xleftrightarrow{b} T(\mathcal{G}_1)$. However, it is guaranteed that there also exists an LTS \mathcal{G}_2 for r with a corresponding match m_2 for which $\mathcal{G}_2 \not\xleftrightarrow{b} T(\mathcal{G}_2)$.

Proposition 3.2 Consider a transformation rule $r = (\mathcal{L}, \mathcal{R})$. Let \mathbb{G} be the set of all LTSs. Let $r_{\mathcal{G}}$ be the set of all possible match pairs corresponding to a transformation of an LTS \mathcal{G} where $r_{\mathcal{G}}$ consists of tuples of the form $(m : \mathcal{L} \rightarrow \mathcal{G}, \hat{m} : \mathcal{R} \rightarrow T(\mathcal{G}))$. The following holds

$$(\forall \mathcal{G} \in \mathbb{G}, (m, \hat{m}) \in r_{\mathcal{G}}. \mathcal{G} \xleftrightarrow{b} T(\mathcal{G})) \Rightarrow \mathcal{L}^{\kappa} \xleftrightarrow{b} \mathcal{R}^{\kappa}$$

Proof Assume that for all $\mathcal{G} \in \mathbb{G}$ and $(m, \hat{m}) \in r_{\mathcal{G}}$ it holds that $\mathcal{G} \xleftrightarrow{b} T(\mathcal{G})$. Trivially, we have $\mathcal{L}^{\kappa} \in \mathbb{G}$ and trivial matches $m : \mathcal{L} \rightarrow \mathcal{L}^{\kappa}, \hat{m} : \mathcal{R} \rightarrow T(\mathcal{L}^{\kappa})$. It follows from the assumption that $\mathcal{L}^{\kappa} \xleftrightarrow{b} T(\mathcal{L}^{\kappa})$. Moreover, by Definition 3.5, $T(\mathcal{L}^{\kappa}) = \mathcal{R}^{\kappa}$. It follows that $\mathcal{L}^{\kappa} \xleftrightarrow{b} \mathcal{R}^{\kappa}$. \square

4. Verifying sets of dependent LTS transformations

In this section, we extend the setting by considering *sets* of interacting process LTSs in so-called *networks of LTSs* [Lan06] or *LTS networks*. Transformations can now affect multiple LTSs in an input network, and the analysis of transformations is more involved, since changes to process-local behaviour may affect system-global properties. Finally, we prove the correctness of the technique based on the complete Coq proof. From the proof it follows that per set of related transformation rules, only a single bisimulation check is required in order to verify a system of transformation rules.

4.1. LTS networks and their transformation

An LTS network (Definition 4.1) describes a system consisting of a finite number of concurrent process LTSs and a set of synchronisation laws which define the possible interaction between the processes. The explicit behaviour of an LTS network is defined by its *system LTS* (Definition 4.2). 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 i^{th} element of vector \bar{v} .

Definition 4.1 (*LTS network*) An LTS network \mathcal{N} of size n is a pair (Π, \mathcal{V}) , where

- Π is a vector of n concurrent LTSs. For each $i \in 1..n$, we write $\Pi_i = (\mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i)$ and $s \xrightarrow{a}_i s'$ as shorthand for $s \xrightarrow{a}_{\Pi_i} s'$.
- \mathcal{V} is a finite set of synchronisation laws. A *synchronisation law* is a tuple (\bar{v}, a) , where \bar{v} is a vector of size n , called the *synchronisation vector*, describing synchronising action labels, and a is an action label representing the result of successful synchronisation. We have $\forall i \in 1..n. \bar{v}_i \in \mathcal{A}_i \cup \{\bullet\}$, where \bullet is a special symbol denoting that Π_i performs no action.

Definition 4.2 (*System LTS*) Given an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$, its *system LTS* is defined by $\mathcal{G}_{\mathcal{N}} = (\mathcal{S}_{\mathcal{N}}, \mathcal{A}_{\mathcal{N}}, \mathcal{T}_{\mathcal{N}}, \mathcal{I}_{\mathcal{N}})$, with

- $\mathcal{A}_{\mathcal{N}} = \{a \mid (\bar{v}, a) \in \mathcal{V}\}$;
- $\mathcal{I}_{\mathcal{N}} = \{\langle s_1, \dots, s_n \rangle \mid s_i \in \mathcal{I}_i\}$, and
- $\mathcal{T}_{\mathcal{N}}$ and $\mathcal{S}_{\mathcal{N}}$ are the smallest relation and set, respectively, satisfying $\mathcal{I}_{\mathcal{N}} \subseteq \mathcal{S}_{\mathcal{N}}$ and for all $\bar{s} \in \mathcal{S}_{\mathcal{N}}, (\bar{v}, a) \in \mathcal{V}$:

$$(\forall i \in 1..n. \bar{v}_i \neq \bullet \Rightarrow \bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i) \Rightarrow \bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}' \wedge \bar{s}' \in \mathcal{S}_{\mathcal{N}},$$

where $\bar{s}'_i = \bar{s}_i$ for all i with $\bar{v}_i = \bullet$.

The system LTS is obtained by combining the transitions of the processes in Π according to the synchronisation laws in \mathcal{V} . Whenever we want to make explicit that a transition $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ is enabled by a synchronisation law (\bar{v}, a) , we write $\bar{s} \xrightarrow{\bar{v}, a}_{\mathcal{N}} \bar{s}'$. We refer to $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ and $\bar{s} \xrightarrow{\bar{v}, a}_{\mathcal{N}} \bar{s}'$ transitions as *global transitions* and we refer to transitions $\bar{s} \xrightarrow{\bar{v}_i}_i \bar{s}'$ ($i \in \text{Ac}(\bar{v})$) as the *(process-)local transitions*. If it is clear from the context whether a transition is global or local, then “global” or “local” is omitted.

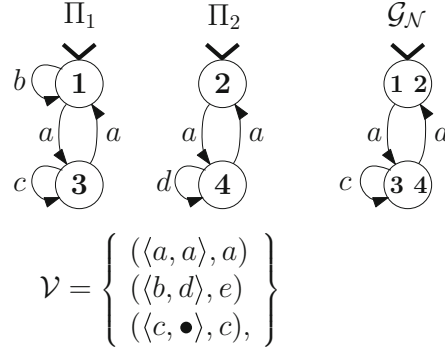


Fig. 6. An LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ (left) and its system LTS $\mathcal{G}_{\mathcal{N}}$ (right)

The LTS network model subsumes most hiding, renaming, cutting, and parallel composition operators present in process algebras, but also more expressive operators such as m among n synchronisation [LM13]. For instance, hiding can be applied by replacing the a component in a law by τ . A transition of a process LTS is *cut* if it is blocked with respect to the behaviour of the whole system (system LTS), i.e., there is no synchronisation law involving the transition's action label at the position of the process LTS.

Figure 6 shows an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ with two processes and three synchronisation laws (left) and its system LTS $\mathcal{G}_{\mathcal{N}}$ (right). To construct the system LTS, first, the initial states of Π_1 and Π_2 are combined to form the initial state of $\mathcal{G}_{\mathcal{N}}$. Then, the outgoing transitions of the initial states of Π_1 and Π_2 are combined using the synchronisation laws, leading to new states in $\mathcal{G}_{\mathcal{N}}$, and so on.

Law $\langle (a, a), a \rangle$ specifies that the process LTSs can synchronise on their a -transitions, resulting in a -transitions in the system LTS. The other laws specify that b - and d -transitions can synchronise, resulting in e -transitions, and that c -transitions can be fired independently. Note that in fact, b - and d -transitions in Π_1 and Π_2 are never able to synchronise.

The set of indices of processes participating in a synchronisation law (\bar{v}, a) is defined as $Ac(\bar{v}) = \{i \mid i \in 1..n \wedge \bar{v}_i \neq \bullet\}$; e.g., $Ac(\langle c, b, \bullet \rangle) = \{1, 2\}$.

Branching bisimilarity is a congruence for construction of the system LTS of LTS networks if the synchronisation laws do not synchronise, rename, or cut τ -transitions [Lan06]. Given an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$, these properties are formalised as follows:

1. $\forall (\bar{v}, a) \in \mathcal{V}, i \in 1..n. \bar{v}_i = \tau \Rightarrow Ac(\bar{v}) = \{i\}$ (no synchronisation of τ 's);
2. $\forall (\bar{v}, a) \in \mathcal{V}, i \in 1..n. \bar{v}_i = \tau \Rightarrow a = \tau$ (no renaming of τ 's);
3. $\forall i \in 1..n. \tau \in \mathcal{A}_i \Rightarrow \exists (\bar{v}, a) \in \mathcal{V}. \bar{v}_i = \tau$ (no cutting of τ 's).

In this article, we only consider LTS networks satisfying these properties.

Transformation of an LTS network. A *system of transformation rules*, or a *rule system* for short, allows the transformation of LTS networks. A rule system transforms multiple processes and may introduce new synchronisation laws. The rule system is defined as follows.

Definition 4.3 (Rule system) A *rule system* $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ consists of a vector of transformation rules R , a set of synchronisation laws \mathcal{V}' that must be present in the networks that Σ is applied on, and a set of synchronisation laws $\hat{\mathcal{V}}$ introduced in the network resulting from a transformation. The i^{th} left and right pattern LTSs of R are denoted by \mathcal{L}_i and \mathcal{R}_i , respectively.

Intuitively, a rule system describes how a concurrent system is modified to create a transformed concurrent system. A rule system is designed with a specific result in mind. Therefore, it is desirable that a rule system is *confluent* such that transformation rules can be applied in any order eventually leading to the same result. Checking confluence can be done efficiently [Wij15]. In the remainder of this article, we only consider confluent rule systems.

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

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

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

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

Definition 4.4 (*Projection of synchronisation vectors*) Let $f : 1..n \rightarrow 1..m$ with $n, m \in \mathbb{N}$ be a partial mapping. For $i \in 1..n$ we write $f(i) = *$ to indicate that i is not mapped by f . A synchronisation vector \bar{v} of size m can be projected iff for all $j \in Ac(\bar{v})$ there exists an $i \in 1..n$ such that $f(i) = j$. This condition ensures that all active indices of \bar{v} are represented in the projected vector. The *projected synchronisation vector*, denoted \bar{v}^f , is a vector of size n with elements $i \in 1..n$ defined as:

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

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

For a vector of matches \bar{M} , the transformation step is formalised in Definition 4.5. The transformed LTS network $T_{\bar{M}}(\mathcal{N})$ consists of the transformed process LTSs and the original set of synchronisation laws \mathcal{V} updated with the projection of $\hat{\mathcal{V}}$.

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

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

Definition 4.5 (*LTS network transformation step*) Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system. Let \bar{M} be a vector of size n of match pairs (m_i, \hat{m}_i) such that $\mathcal{V}'^{\bar{M}} \subseteq \mathcal{V}$. For all $i \in 1..n$ let T_{m_i} denote the LTS transformation of Π_i using application matches m_i and \hat{m}_i according to Definition 3.5.

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

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

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

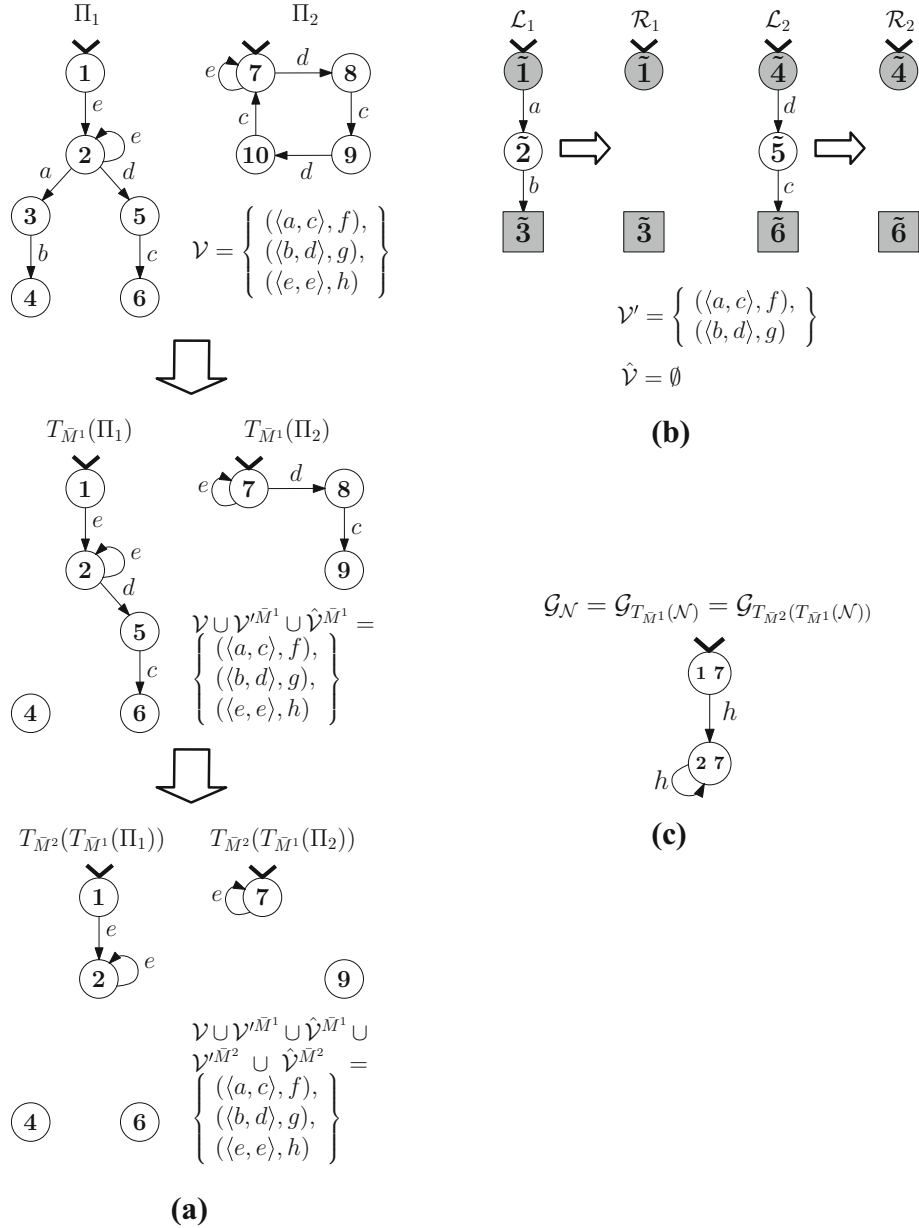


Fig. 7. Exhaustive application of Σ on a network $\mathcal{N} = (\Pi, \mathcal{V})$ where \bar{M}^1 contains the left matches $m_1^1 : \mathcal{L}_1 \rightarrow \Pi_1$ and $m_2^1 : \mathcal{L}_2 \rightarrow \Pi_2$ with $m_1^1 = \{\tilde{1} \mapsto 2, \tilde{2} \mapsto 3, \tilde{3} \mapsto 4\}$ and $m_2^1 = \{\tilde{4} \mapsto 9, \tilde{5} \mapsto 10, \tilde{6} \mapsto 7\}$, and \bar{M}^2 contains the left matches $m_1^2 : \mathcal{L}_2 \rightarrow \Pi_1$ and $m_2^2 : \mathcal{L}_2 \rightarrow \Pi_2$ with $m_1^2 = \{\tilde{4} \mapsto 2, \tilde{5} \mapsto 5, \tilde{6} \mapsto 6\}$ and $m_2^2 = \{\tilde{4} \mapsto 7, \tilde{5} \mapsto 8, \tilde{6} \mapsto 9\}$. **a** A transformation sequence resulting from the application of Σ on $\mathcal{N} = (\Pi, \mathcal{V})$ with match pair vectors \bar{M}^1 and \bar{M}^2 . **b** Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ removes the a -, b -transition sequence, and the d -, c -transition sequence; the system LTS of networks Σ is applied on remain unchanged as the a - and d -transitions can both never synchronise and the b - and c -transitions are therefore unreachable. **c** The system LTSs of the networks in the transformation sequence; since the a - and c -transitions and the b - and d -transitions can never synchronise and the d - and c -transitions in Π_1 are cut, the h -transitions are the only reachable behaviour.

Figure 7 presents a transformation sequence that is the result of the application of a rule system Σ (see Fig. 7b) on a network \mathcal{N} . The system LTSs of the input and output networks are exactly the same, this LTS is shown in Fig. 7c. The a - and c -transitions and the b - and d -transitions can never synchronise. Furthermore, the d - and c -transitions in Π_1 are cut. The synchronisation of the e -transitions, resulting in h -transitions, are the only reachable behaviour in the system LTS.

Rule system Σ removes the a -, b -transition sequence, and the d -, c -transition sequence. The system LTS of the network Σ is applied on remains unchanged because in the described situation synchronisation of both the a - and d -transitions is impossible and the b - and c -transitions are otherwise unreachable. Transformations like this are useful to gain insights in the reachable behaviour of local process LTSs.

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

4.2. Analysing transformations of an LTS network

In a rule system, transformation rules can be dependent on each other regarding the behaviour they affect. In particular, the rules may refer to actions that require synchronisation according to some law, either in the network being transformed, or the network resulting from the transformation. Since in general, it is not known a priori whether or not those synchronisations can actually happen (see Fig. 6, the a -transitions versus the b - and d -transitions), full analysis of such rules must consider both successful and unsuccessful synchronisation.

To achieve this, dependent rules must be analysed together as combinations of LTS patterns, as shown in Fig. 1. To this end, LTS patterns are combined into an LTS network, called a *pattern network* $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$, with $\bar{\Phi}$ a vector of pattern LTSs, and \mathcal{W} a set of synchronisation laws. In particular, the left and right pattern networks of a rule system $\Sigma = (R, \mathcal{V}, \hat{\mathcal{Y}})$ are defined as $\bar{\mathcal{L}} = (\langle \mathcal{L}_1, \dots, \mathcal{L}_{|R|} \rangle, \mathcal{V})$ and $\bar{\mathcal{R}} = (\langle \mathcal{R}_1, \dots, \mathcal{R}_{|R|} \rangle, \mathcal{V} \cup \hat{\mathcal{Y}})$. For the analysis of these pattern networks, we define in Definition 4.6 the κ -extended pattern network consisting of the combination of the κ -extended LTS patterns and an extension of the synchronisation laws with κ -synchronisation laws \mathcal{W}^κ . The left and right κ -extended pattern networks are denoted $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ and, for the purpose of equivalence checking, must use the same set of κ -synchronisation laws \mathcal{W}^κ .

Definition 4.6 (κ -Extended Pattern Network) Given a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n , its κ -extended pattern network is defined as \mathcal{P}^κ , where

$$\begin{aligned} \mathcal{P}^\kappa &= (\langle \bar{\Phi}_1^\kappa, \dots, \bar{\Phi}_n^\kappa \rangle, \mathcal{W} \cup \mathcal{W}^\kappa), \text{ and} \\ \mathcal{W}^\kappa &= \{(\bar{v}, \mu) \mid Ac(\bar{v}) \neq \emptyset \wedge \forall i \in Ac(\bar{v}). \\ &\quad ((\exists p \in \mathcal{I}_{\bar{\Phi}_i}. \bar{v}_i = \sigma_p) \vee (\exists p \in \mathcal{E}_{\bar{\Phi}_i}. \bar{v}_i = \varepsilon_p) \vee (\exists p \in \mathcal{E}_{\bar{\Phi}_i}. p' \in \mathcal{I}_{\bar{\Phi}_i}. \bar{v}_i = \gamma_{p,p'}))\} \end{aligned}$$

with μ an action that is unique w.r.t. rule system $\Sigma = (R, \mathcal{V}, \hat{\mathcal{Y}})$, i.e., $\mu \neq \tau \wedge \forall (\bar{v}', a) \in \mathcal{V} \cup \hat{\mathcal{Y}}. \mu \neq a$.

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

Figure 8 shows a rule system Σ , in which the two rules are dependent. Again, the states are numbered such that matches can be identified by the state label, i.e., a state \tilde{i} is matched onto state i . The two transformation rules depicted in Fig. 8a introduce a new dependency between two (possibly) independent systems. The corresponding κ -extended pattern networks are given in Fig. 8b. The κ -synchronisation laws allow σ -, ε -, and γ -actions to be performed both independently and synchronised. The synchronisations of σ_1 - and σ_2 -transitions, σ_1 - and ε_2 -transitions, and σ_1 - and $\gamma_{2,2}$ -transitions are displayed as the $\sigma\sigma$ -transition, the $\sigma\varepsilon$ -transition, and the $\sigma\gamma$ -transition, respectively. Figure 8c presents the branching bisimulation check performed on the two κ -extended pattern networks.

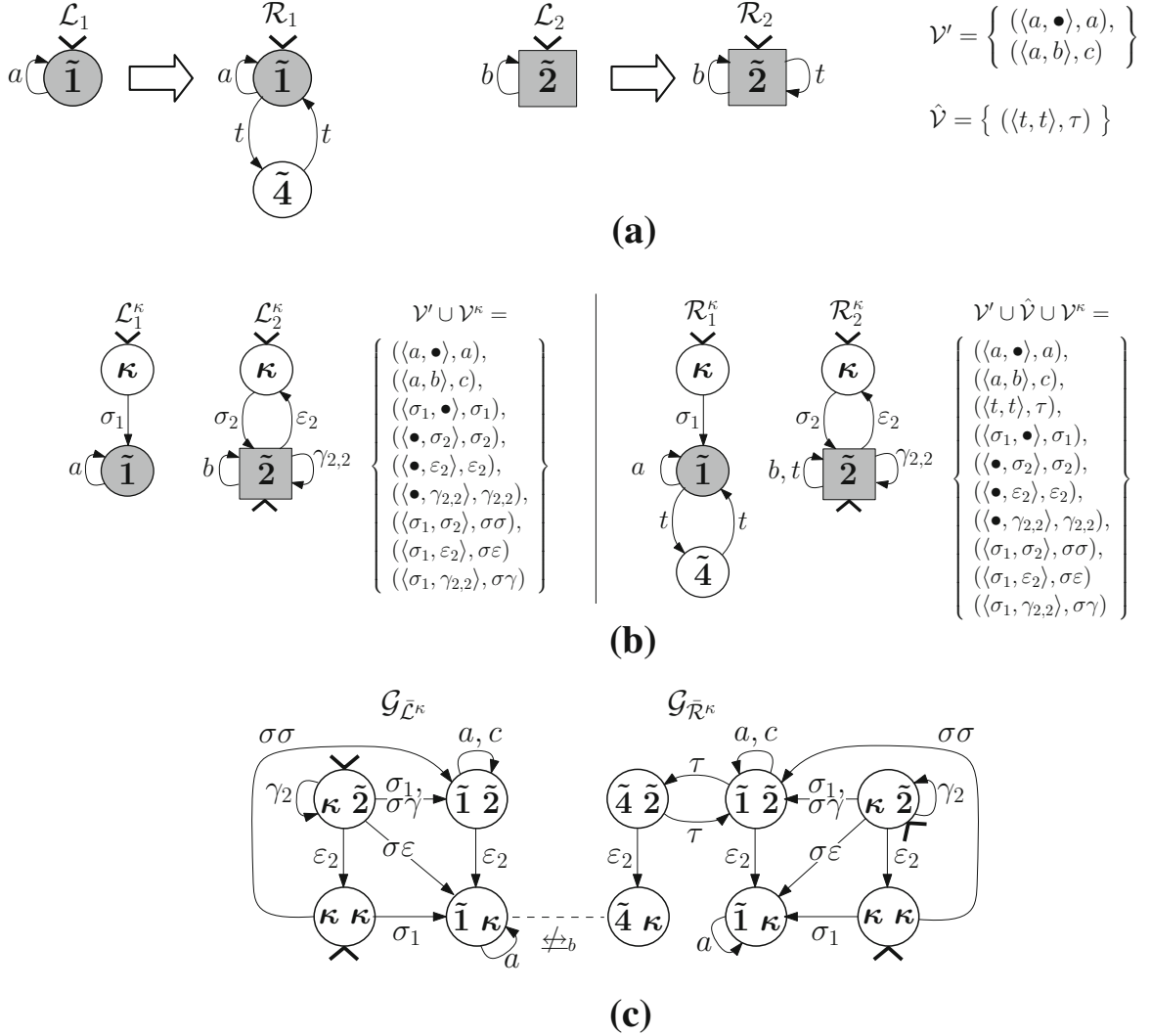


Fig. 8. A rule system and its κ -extended pattern networks and bisimulation checks. **a** A rule system $\Sigma = (R, \mathcal{V}, \hat{\mathcal{V}})$ **b** The corresponding κ -extended pattern networks **c** Bisimulation check; the ε_2 -transition ensures that $\mathcal{L}^\kappa \not\sim_b \mathcal{R}^\kappa$

The check concludes that the two networks are not branching bisimilar. In particular, when the second process (\mathcal{R}_2^κ) leaves the pattern LTS at state $\langle \tilde{4}, \tilde{2} \rangle$ via the ε_2 -transition, the a -transition can no longer be mimicked. The same would occur in any application of Σ at any state matched by $\langle \tilde{4}, \tilde{2} \rangle$ that has a transition to an unmatched state.⁵

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

⁵ In our previous work [PW16], this type of rule system was called a non-cascading rule system. In this work, we no longer need to verify whether a rule system is cascading since the κ -state makes the effect of the transformation on the unmatched states explicit. Furthermore, the correctness proof of the technique presented in this article does not distinguish between cascading and non-cascading rule systems. Hence, the verification technique as described here will correctly reject non-cascading rule systems.

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

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

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

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

Definition 4.7 (*Projection of a rule system*) Let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system with a partition \mathbb{D} of dependency sets. The projection of Σ along a dependency set $P \in \mathbb{D}$ is a rule system $\Sigma^P = (R^P, \mathcal{V}^P, \hat{\mathcal{V}}^P)$ with R^P a vector of size $|P|$ such that for all $i \in |P|$ we have $R_i^P = R_{P(i)}$.

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

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

In Sections 4.2.1 and 4.2.2 the analysis of a rule system and application of a rule system, respectively, is discussed further. The analysis of a rule system consists of the verification of the pattern networks and the analysis condition. The analysis of the application of a rule system constitutes the verification of the application conditions. Both sections present an analysis algorithm and a time complexity analysis.

4.2.1. Analysis of a rule system

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

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

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

Figure 9 shows how the application of a rule system that does not satisfy ANCI affects the transformation verification. The rule system Σ , shown in Fig. 9a, has a synchronisation law $((a, a), a) \in \mathcal{V}'$. However, transformation rule R_2 does not contain any a -transitions, i.e., ANCI is not satisfied. As a result the effects of the transformation of the a -transition by rule R_1 is not visible in the κ -extended pattern networks presented in Fig. 9b. Figure 9d shows that an input network \mathcal{N} exists (Fig. 9c) such that the input network \mathcal{N} and output network $T_\Sigma(\mathcal{N})$ are not branching bisimilar. If rule R_2 would contain the a -loop, then the a -transition would not have been cut and $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ would not be bisimilar any longer. Hence, it is vital that labels considered by synchronisation laws in \mathcal{V}' is also present in the transformation rules, i.e., rule systems must adhere to ANCI.

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

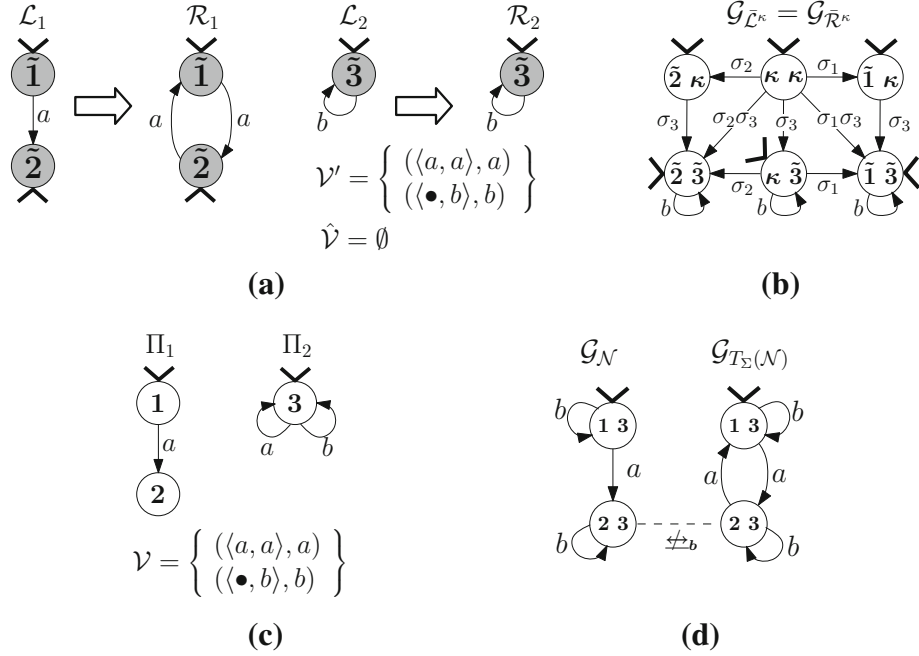


Fig. 9. Rule system Σ does not satisfy **ANCI**: although $\tilde{\mathcal{L}}^{\kappa} \leftrightarrow_b \tilde{\mathcal{R}}^{\kappa}$, a network \mathcal{N} exists such that $\mathcal{G}_{\mathcal{N}} \not\leftrightarrow_b \mathcal{G}_{T_{\Sigma}(\mathcal{N})}$. **a** A rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ that does not satisfy **ANCI**. **b** $\tilde{\mathcal{L}}^{\kappa}$ and $\tilde{\mathcal{R}}^{\kappa}$ are equivalent since a -transitions are not considered by rule 2. **c** An input network $\mathcal{N} = (\Pi, \mathcal{V})$. **d** The system LTSs before (\mathcal{N}) and after ($T_{\Sigma}(\mathcal{N})$) transformation are not branching bisimilar

1. Check whether in Σ , no τ -transitions can be synchronised, renamed, or cut, and whether **ANCI** is satisfied. If not, report which check failed and stop.
2. Check whether the rules in Σ are confluent. If not, report that Σ is not confluent and stop.
3. For each rule in R the κ -extended pattern LTSs are constructed according to Definition 3.7.
4. Construct the set of dependency sets \mathbb{D} .
5. For each class (dependency set) $P \in \mathbb{D}$ determine whether $\tilde{\mathcal{L}}^{\kappa, P} \leftrightarrow_b \tilde{\mathcal{R}}^{\kappa, P}$ holds, i.e., whether the κ -extended pattern networks projected along P are branching bisimilar.

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

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

In the *second* step, it is checked whether Σ is confluent. Confluence checking of transformations of LTSs has $O(\binom{|R|}{2} \cdot s^2 \cdot t \cdot \log(s))$ time complexity [Wij15], with s and t the largest number of states and transitions in an LTS pattern of a rule in Σ , respectively.

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

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

In the *fifth* and last step, for each dependency set $P \in \mathbb{D}$ the pattern networks $\tilde{\mathcal{L}}^{\kappa, P}$ and $\tilde{\mathcal{R}}^{\kappa, P}$ are constructed and it is verified whether $\tilde{\mathcal{L}}^{\kappa, P}$ and $\tilde{\mathcal{R}}^{\kappa, P}$ are branching bisimilar. Hence, $|\mathbb{D}|$ bisimulation checks are performed. Let s , t and a be the largest number of states, transitions and action labels, respectively, appearing in the κ -

extended pattern networks of Σ . Branching bisimilarity checking can be performed in $O(t \cdot \log(s + a))$ [GW16]. Therefore, the time complexity of the final step of the analysis is $O(|\mathbb{D}| \cdot (t \cdot \log(s + a)))$.

The running time of steps 3-5 together therefore amounts to $O(|\mathbb{D}| \cdot (t \cdot \log(s + a)) + |\mathcal{V} \cup \hat{\mathcal{V}}| + |R| \cdot g)$. In contrast with previous work, the analysis presented here only requires a single bisimulation check per dependency set $P \in \mathbb{D}$ (versus $2^{|\mathbb{D}|} - 1$ in previous work [WE13]). This improvement is made possible by the new correctness proof presented in Sect. 4.3.

4.2.2. Analysis of the application of a rule system

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

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

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

$$\begin{aligned} \forall j \in 1..|R|, (\bar{v}, a) \in \mathcal{V}'. \{j\} \subset Ac(\bar{v}) \wedge \bar{v}_j \in \mathcal{A}_{\mathcal{L}_j} \Rightarrow \\ \forall i \in 1..n, (s, \bar{v}_j, s') \in \mathcal{I}_i. \exists (m:\mathcal{L}_j \rightarrow \Pi_i, _) \in \mathbf{m}, (p, \bar{v}_j, p') \in \mathcal{I}_{\mathcal{L}_j}. m(p) = s \wedge m(p') = s' \end{aligned} \quad (\text{APC1})$$

We write “ $_$ ” to indicate that the second element of the match pair is not relevant. The symmetric condition involving the \mathcal{R}_j and $\mathcal{V} \cup \hat{\mathcal{V}}$ applies as well. Together with [ANCI](#), [APC1](#) ensures that synchronising transitions with a particular label in the input network are either all transformed, or none are transformed. This is shown in Sect. 4.3 in Lemma 4.9.

Figure 10 shows a transformation that satisfies [ANCI](#), but does not adhere to [APC1](#). The rule system Σ , presented in Fig. 10a, transforms a -transitions to c -transitions. The first transformation rule transforms an a -transition to a c -transition iff there is a b -loop at the state from which the a -transition is performed. The second transformation rule transforms a -loops to c -loops. If Σ is applied on network \mathcal{N} , presented in Fig. 10c, then the transition $\langle 1 \rangle \xrightarrow{a}_{\Pi_1} \langle 2 \rangle$ is not transformed. Therefore, the transformation does not satisfy [APC1](#).

The laws of the rule system describe that the synchronisation of two a -transitions results in an a -transition (i.e., $((a, a), a) \in \mathcal{V}'$), and that the b -loop is performed independently of other processes (i.e., $((b, \bullet), b) \in \mathcal{V}'$). A new synchronisation law $((c, c), a)$ is added such that the synchronisation of two c -transitions results in an a -transition again. This makes old and new synchronising behaviour comparable. As shown in Fig. 10b, the branching bisimulation check cannot distinguish between the left and right κ -extended pattern networks.

However, if Σ is applied on the LTS network \mathcal{N} given in Fig. 10c, then it turns out that $\mathcal{G}_{\mathcal{N}}$ and $\mathcal{G}_{T_{\Sigma}(\mathcal{N})}$ are not branching bisimilar (see Fig. 10d). The transformed network can no longer perform the $\langle 2, 3 \rangle \xrightarrow{a} \langle 1, 3 \rangle$ transition. Transition $\langle 2 \rangle \xrightarrow{a} \langle 1 \rangle$ in process Π_1 has not been transformed while the a -loop in Π_2 has been transformed to a c -loop in $T(\Pi_2)$. Hence, there is no a -transition available anymore with which the $\langle 2 \rangle \xrightarrow{a}_{T(\Pi_2)} \langle 1 \rangle$ transition can synchronise.

The *second* condition prevents that projections of new synchronisation laws in $\hat{\mathcal{V}}$ are defined over actions already present in the processes of an input network. Otherwise, an LTS network could be altered without actually defining any transformation rules. Formally, if the left LTS pattern \mathcal{L}_j ($j \in |R|$) of the j^{th} rule in R is matched on the i^{th} process ($i \in 1..n$) in Π , then the \bar{v}_j may not be defined over actions in \mathcal{A}_i .

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

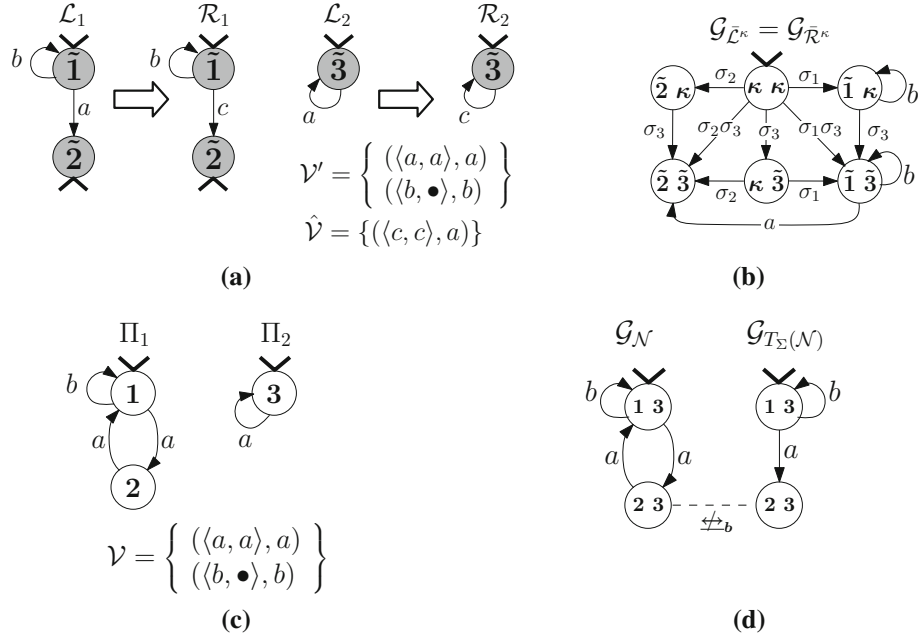


Fig. 10. Rule system Σ and input network \mathcal{N} with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy **APC1**; although $\tilde{\mathcal{L}}^\kappa \xleftrightarrow{b} \tilde{\mathcal{R}}^\kappa$, the system LTS of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$. **a** Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ transforms a -transitions to c -transitions, synchronisation of c -transitions results in an a -transition. **b** $\tilde{\mathcal{L}}^\kappa$ and $\tilde{\mathcal{R}}^\kappa$ are equivalent since the synchronisation of two c -transitions result in an a -transition again. **c** An input network $\mathcal{N} = (\Pi, \mathcal{V})$. **d** The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; the transformed model is no longer able to synchronise the a -transition performed at state (2) that was not transformed, since the loop at state (3) has been transformed to a c -loop

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

The *third* and *fourth* condition concern how the set of laws \mathcal{V} (of the network \mathcal{N} on which Σ is applied) is related to the set of laws \mathcal{V}' (that Σ expects). Consider a set of match pairs m describing the transformation of \mathcal{N} as defined by Σ . The application of the matches in m is distributed over a sequence of transformation steps. Let \bar{M} be a vector of match pairs defining a single transformation step in the sequence. For each transformation step \bar{M} it is required that **APC3** and **APC4** hold.

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

$$\mathcal{V}'^{\bar{M}} \subseteq \mathcal{V} \quad (\text{APC3})$$

An application of a rule system Σ on an LTS network \mathcal{N} for which **APC3** does not hold is given in Fig. 11. The corresponding match pair vector is $\bar{M} = (m : \mathcal{L}_1 \rightarrow \Pi_1, \hat{m} : \mathcal{R}_1 \rightarrow T(\Pi_1))$. Condition **APC3** does not hold since the law $\{(a), c\} \in \mathcal{V}'$ of rule system Σ , presented in Fig. 11a is not included in the set of laws of the input network \mathcal{N} , shown in Fig. 11c. The analysis condition **ANCI** and application conditions **APC1** and **APC2** hold.

The rule system transforms a -transition to d -transitions. The local a -transitions result in global c -transitions due to law $\{(a), c\} \in \mathcal{V}'$. To ensure that the behaviour remains equivalent a new synchronisation law $\{(d), c\} \in \hat{\mathcal{V}}$ is introduced such that, like the a -transitions, the d -transitions result in global c -transitions. As shown in Fig. 11b the left and right κ -extended pattern networks are branching bisimilar, as expected.

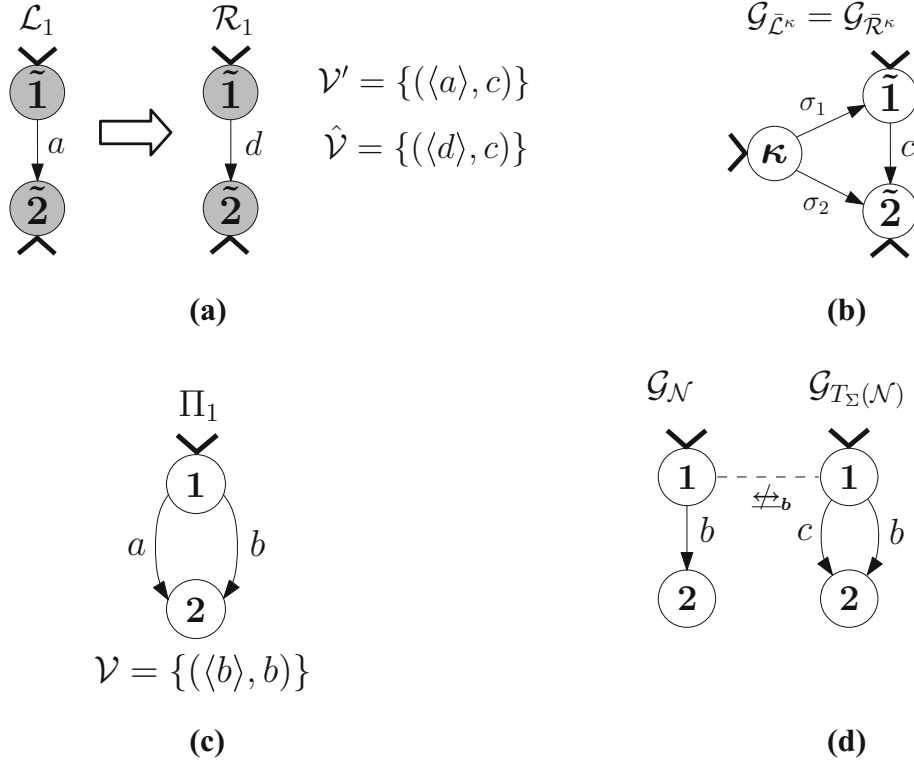


Fig. 11. Rule system Σ and input network \mathcal{N} with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy **APC3**; although $\tilde{\mathcal{L}}^\kappa \leftrightarrow_b \tilde{\mathcal{R}}^\kappa$, the system LTSs of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$. **a** Rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ transforms a -transitions to d -transitions, the synchronisation laws specify that both the local a - and the d -transitions result in a global c -transition. **b** $\tilde{\mathcal{L}}^\kappa$ and $\tilde{\mathcal{R}}^\kappa$ are equivalent since for both the left and right patterns the process-local transitions results in a c -transition. **c** Input network $\mathcal{N} = (\Pi, \mathcal{V})$. **d** The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; since $((a), c) \in \mathcal{V}' \setminus \mathcal{V}$ (i.e., **APC3** is violated) the a -transition is cut in Π_1 while the d -transition is not cut in $T(\Pi_1)$ due to introduction of $((d), c)$

However, when Σ is applied on input network \mathcal{N} the transformation of the a -transition in process Π_1 (now a d -transition) is not cut due to introduction of the law $((d), c) \in \hat{\mathcal{V}}$. The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are given in Fig. 11d. Since $\mathcal{V}' \not\subseteq \mathcal{V}$, an analysis of Σ does not take into account that in Π_1 the a -transition is cut. Therefore, the analysis cannot give any guarantees for the input network \mathcal{N} .

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

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

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

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

A transformation that does not satisfy **APC4** is presented in Fig. 12. Condition **APC4** is not satisfied because the law $((a, c), e) \in \mathcal{V} \setminus \mathcal{V}'$ of input network \mathcal{N} , shown in Fig. 12c, contains behaviour that influences the transformation rules of rule system Σ , shown in Fig. 12a. The transformation satisfies conditions **ANCI1**, **APC1**, **APC2**, and **APC3**.

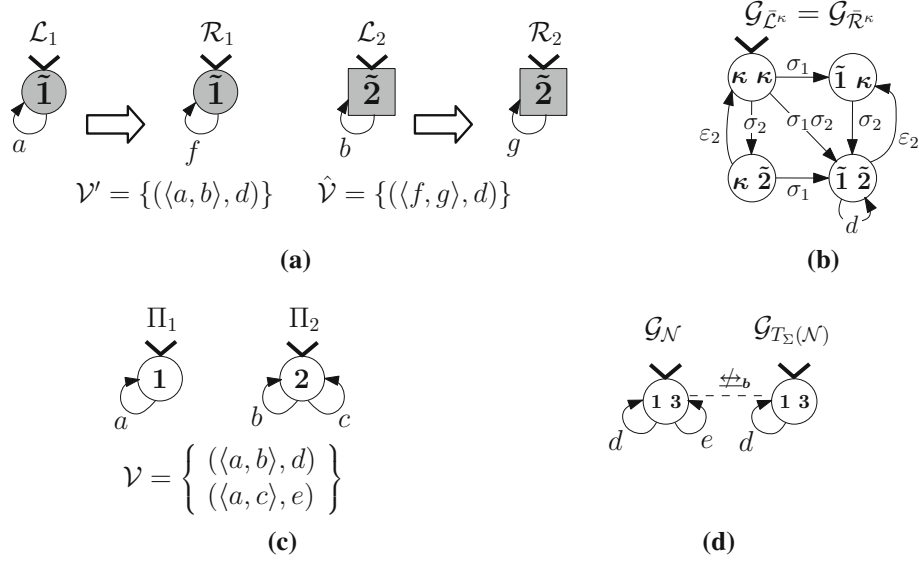


Fig. 12. Rule system Σ and input network \mathcal{N} with matches $m(\tilde{i}) = i$ and $\hat{m}(i) = i$ that do not satisfy **APC4**; although $\tilde{\mathcal{L}}^\kappa \leftrightarrow_b \tilde{\mathcal{R}}^\kappa$, the system LTSs of the input network $\mathcal{G}_{\mathcal{N}}$ is *not* bisimilar to system LTS of the output network $\mathcal{G}_{T_\Sigma(\mathcal{N})}$. **a** Rule system $\Sigma = (R, \mathcal{V}, \hat{\mathcal{V}})$ transforms a -loops to f -loops and b -loops to g -loops, like synchronisation of a and b labels, the synchronisation of f and g labels results in an d -transition. **b** $\tilde{\mathcal{L}}^\kappa$ and $\tilde{\mathcal{R}}^\kappa$ are equivalent since the transformed loops synchronisation to a d -loop again. **c** Input network $\mathcal{N} = (\Pi, \mathcal{V})$. **d** The system LTSs before ($\mathcal{G}_{\mathcal{N}}$) and after ($\mathcal{G}_{T_\Sigma(\mathcal{N})}$) transformation are not branching bisimilar; the transformed model is unable to synchronise the b -transition of Π_2 that was not transformed because the a -loop in Π_1 has been transformed to an f -loop.

Rule system Σ transforms a -loops to f -loops and b -loops to g -loops. In an attempt to preserve the semantics the f - and g -actions, like the a - and b -actions, they are forced to synchronise, resulting in d -actions. As a result the left and right κ -extended pattern networks, presented in Fig. 12b, are branching bisimilar.

However, if Σ is applied on network \mathcal{N} , then the possibility of synchronising the a - and c -loops is lost. It follows that $\mathcal{G}_{\mathcal{N}}$ can perform an e -loop while $\mathcal{G}_{T_\Sigma(\mathcal{N})}$ cannot (see Fig. 12d). Hence, the two system LTSs are not branching bisimilar.

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

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

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

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

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

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

In the *first* step, the maximum set of matches is calculated. In general, the graph matching problem [DP06] is NP-complete. However, it has been shown in [DP06] that if the graphs have a root, all states are reachable from that root, and each state has a bounded number b of outgoing transitions, then the complexity is independent of the size of the input graph, instead only depending on b and the number of transitions n in the left pattern of the transformation rule. The complexity is then $O(\sum_{i=0}^n b_i)$. Since our LTSs are weakly connected, they meet these requirements.

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

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

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

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

4.3. Correctness of the verification

In this section, we prove the correctness of the rule system verification as described in the previous section. We prove the soundness of the rule system verification in Proposition 4.1. The completeness of the verification approach is shown in Proposition 4.2.

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

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

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

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

The κ -extended pattern networks can be seen as an abstraction from the input networks. In a κ -extended pattern network, individual processes can only leave the κ -state via σ -transitions and only enter the κ -state via ε -transitions. Hence, for all transitions enabled by laws in the original (non- κ -extended) pattern network, the processes that are in the κ -state before such a transition is executed are still in the κ -state after the transition has been followed. This property is formalised in Lemmas 4.1 and 4.2.

Lemma 4.1 Consider a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n . Then,

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

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

Say $\bar{p}_i = \kappa$. By Definition 3.7, σ -transitions can only be performed from a κ -state. Similarly, a process can only enter a κ -state by performing an ε -transition. Hence, since $(\bar{v}, a) \in \mathcal{W}$ and $\bar{p}_i = \kappa$, we must have $\bar{v}_i = \bullet$. It follows from Definition 4.2 that $\bar{p}_i = \bar{p}'_i$. Hence, we have $\bar{p}'_i = \kappa$.

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

Lemma 4.1 can be applied inductively to obtain a similar result for τ -paths. Synchronisation laws with τ as a result action are, by Definition 4.6, never κ -synchronisation laws. Therefore, every process that is in a κ -state before a sequence of τ -transitions is still in the κ -state after the sequence of τ -transitions as shown by Lemma 4.2.

Lemma 4.2 Consider a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n . Then,

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

Proof Consider states $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ such that $\bar{p} \xrightarrow{\tau}_{\mathcal{P}^\kappa}^* \bar{p}'$. The use of τ -actions is not allowed in laws in \mathcal{W}^κ , hence it follows that $(\bar{v}, a) \in \mathcal{W}$. Therefore, we have $\bar{p} \xrightarrow{\tau}_{\mathcal{P}}^* \bar{p}'$. The remainder of the proof follows directly from Lemma 4.1 and structural induction on $\bar{p} \xrightarrow{\tau}_{\mathcal{P}}^* \bar{p}'$. \square

Due to the κ -laws, branching bisimulation relations between $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$ preserve κ -states, i.e., when two state vectors are related, any κ -states present in one vector are also present in the other vector at the same positions, and vice versa. This is expressed in Lemma 4.3.

Lemma 4.3 Consider two pattern networks $\bar{\mathcal{L}}$ and $\bar{\mathcal{R}}$ of size n . Then,

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

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

Say $\bar{p}_i = \kappa$. By Definition 3.1, there is at least one state $\hat{p} \in \mathcal{I}_{\mathcal{L}_i}$, and furthermore, according to Definition 3.7, there is a transition $\kappa \xrightarrow{\sigma_{\hat{p}}}_{\mathcal{L}_i} \hat{p}$. Hence, there is a law $(\bar{v}, \mu) \in \mathcal{V}^\kappa$, with $\bar{v}_i = \sigma_{\hat{p}}$ and $\forall j \in 1..n. j \neq i \Rightarrow \bar{v}_j = \bullet$, enabling transition $\bar{p} \xrightarrow{\mu}_{\bar{\mathcal{L}}^\kappa} \bar{p}'$ for some \bar{p}' with $\bar{p}'_i = \hat{p}$ (by Definition 4.6). Since $\bar{p} \leftrightarrow_b \bar{q}$ and $\mu \neq \tau$, we have $\bar{q} \xrightarrow{\tau}_{\bar{\mathcal{R}}^\kappa}^* \hat{q} \xrightarrow{\mu} \bar{q}'$. It follows that $\hat{q}_i \xrightarrow{\sigma_{\hat{p}}} \bar{q}'_i$. Since σ -transitions can only be performed from κ -states, we have $\hat{q}_i = \kappa$. Finally, from Lemma 4.2 it follows that $\bar{q}_i = \kappa$.

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

As κ -extended pattern networks form an abstraction from the matched input network, it is desirable that those states representing states not removed by the transformation are related to themselves. In the κ -extended left and right pattern networks the glue-states and the κ -states represent the states that are kept. As shown in Lemma 4.4, this can be directly lifted to the network-global level when state vectors only contain exit- and κ -states. However, this cannot be guaranteed for state vectors that also contain in-states due to the lack of a unique transition (such as the σ -, ε , and γ -transitions) leaving those in-states. If a state vector \bar{p} consists of in-, out- and κ -states, then \bar{p} may be related to a different state vector \bar{q} via a τ -path originating from an in-state \bar{p}_i contained in \bar{p} . When matches on initial states are restricted to exit-states, Lemma 4.4 is sufficient to show that initial states of the input and output networks of a transformation are related.

Lemma 4.4 Consider a rule system $\Sigma = (R, \mathcal{V}, \hat{\mathcal{V}})$ such that $\bar{\mathcal{L}}^\kappa \leftrightarrow_b \bar{\mathcal{R}}^\kappa$. Then, $\forall \bar{p} \in \mathcal{I}_{\bar{\mathcal{L}}^\kappa}. \bar{p} \leftrightarrow_b \bar{p}$.

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

$$(\bar{v}_i^\kappa, \bar{p}'_i) = \begin{cases} (\varepsilon_{\bar{p}_i}, \kappa) & \text{if } \bar{p}_i \in \mathcal{E}_{\bar{\Phi}_i} \\ (\sigma_x, x) & \text{if } \bar{p}_i = \kappa. \text{ By Definition 3.1, there exists an } x \in \mathcal{I}_{\bar{\mathcal{L}}_i} \end{cases}$$

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

By Definition 3.6, there is a state $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^\kappa}$ such that $\bar{p} \leftrightarrow_b \bar{q}$. Furthermore, since $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$, we have $\mu \neq \tau$. Hence, there is a $\bar{q} \xrightarrow{\tau}_{\bar{\mathcal{R}}^\kappa}^* \hat{q} \xrightarrow{\mu} \bar{q}'$ such that $\bar{p} \leftrightarrow_b \hat{q}$ and $\bar{p}' \leftrightarrow_b \bar{q}'$. We show that $\bar{p} = \hat{q}$ from which it

follows that $\bar{p} \leftrightarrow_b \bar{p}$. Consider an $i \in 1..n$. The σ -transitions only leave from κ -states (in which case $\bar{p}_i = \kappa$ and the $\varepsilon_{\bar{p}_i}$ -transitions only leave from the state \bar{p}_i , i.e., each of the $\bar{p}_i \xrightarrow{\bar{v}_i^\kappa} \bar{p}'_i$ transitions carries a unique label identifying the states connected by the transition. Both \bar{p}_i and \hat{q}_i can perform the $\xrightarrow{\bar{v}_i^\kappa}$ directly. It follows that $\bar{p}_i = \hat{q}_i$. Therefore, $\bar{p} = \hat{q}$ and $\bar{p} \leftrightarrow_b \hat{q}$ can be rewritten to $\bar{p} \leftrightarrow_b \bar{p}$. \square

To formally define how a κ -extended network relates to an input network, we introduce the mapping of state vectors as presented in Definition 4.8. Similar to matches for a single rule, the mapping of a state vector of a κ -extended pattern network defines how a state vector of the pattern is mapped to a state vector of an LTS network.

Definition 4.8 (*State vector mapping*) Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with corresponding matches $m_i : \bar{\Phi}_i \rightarrow \Pi_i$ for all $i \in 1..n$. We say a state vector $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$ is *mapped* to a state vector $\bar{s} \in \mathcal{S}_{\mathcal{N}}$, denoted by $\bar{p} \vdash \bar{s}$, iff

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

By referring to matches of the individual vector elements, a state vector is mapped on to another state vector. Since the κ -state represents unmatched states, the mapping relates the κ -state to all unmatched states. Hence, for every state $\bar{s} \in \mathcal{S}_{\mathcal{N}}$ there is a state $\bar{p} \in \mathcal{P}^\kappa$ that maps on state \bar{s} (Lemma 4.5).

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

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

The state vector mapping preserves the branching structure of the pattern network. Similar to a match, the state vector mapping (Definition 4.8) preserves the branching structure of the pattern network for the set of matching laws. Before proving this claim, we first show that the state vector mapping is complete, i.e., the mapping relation maps to all states of any input network. More precisely, for each (vector) state \bar{s} in the input network there is a (vector) state in the κ -extended pattern network that is mapped on \bar{s} . This is formally proven in Lemma 4.5.

Lemma 4.5 Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ and corresponding matches $m_i : \bar{\Phi}_i \rightarrow \Pi_i$ for all $i \in 1..n$. Then, $\forall \bar{s} \in \mathcal{S}_{\mathcal{N}}. \exists \bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}. \bar{p} \vdash \bar{s}$.

Proof Let \bar{s} be a state in $\mathcal{S}_{\mathcal{N}}$. From the definition of state vector mapping (Definition 4.8) it follows that there is a $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$ with $\bar{p} \vdash \bar{s}$. We shall construct \bar{p} such that $\bar{p} \vdash \bar{s}$. Consider an $i \in 1..n$. If $\exists x \in \mathcal{S}_{\bar{\Phi}_i}. m_i(x) = \bar{s}_i$, then we take $\bar{p}_i = x$. Otherwise, we take $\bar{p}_i = \kappa$. By construction, it holds that $\bar{p} \vdash \bar{s}$. Finally, by Definition 3.7, $x, \kappa \in \mathcal{S}_{\bar{\Phi}_i}$, and thus, $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$. \square

Lemmas 4.6 and 4.7 express that the state vector mapping preserves the branching structure of the pattern network. Lemma 4.6 states that for each transition in the pattern network there is a corresponding transition in the mapped network. Lemma 4.7 extends this to sequences of τ -transitions. In Lemma 4.6 and Lemma 4.7, the end state of the transition and the sequence of transitions in the input network, respectively, is identified. In short, for vector states \bar{p} and \bar{s} , if $\bar{p} \vdash \bar{s}$, then for each index i , firstly, transitions taken in pattern $\bar{\Phi}_i$ can be mimicked by transitions in process Π_i from the mapped state leading to a state mapped by the target state in $\bar{\Phi}_i$, and secondly, if the state \bar{p}_i is a κ -state, then no transitions from the corresponding state \bar{s}_i are taken.

Lemma 4.6 Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ (APC3) and corresponding matches $m_i : \bar{\Phi}_i \rightarrow \Pi_i$ for all $i \in 1..n$. Then,

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

Proof Consider synchronisation law $(\bar{v}, a) \in \mathcal{W}$ and states $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ such that $\bar{p} \xrightarrow{\bar{v}, a}_{\mathcal{P}^\kappa} \bar{p}'$, and $\bar{s} \in \mathcal{S}_{\mathcal{N}}$ with $\bar{p} \vdash \bar{s}$. Take $\bar{s}' := \bar{s}[m_i(\bar{p}'_i) \mid \bar{p}'_i \neq \kappa]$. By construction, we have $\bar{s}' \in \mathcal{S}_{\mathcal{N}}$ and $\forall i \in 1..n. \bar{p}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{s}_i$. Furthermore, Lemma 4.1 ensures that $\bar{p}_i = \kappa \iff \bar{p}'_i = \kappa$ for all $i \in 1..n$. Therefore, for all $i \in 1..n$, it holds that $(\neg \exists x. m_i(x) = \bar{s}_i) \iff (\neg \exists x. m_i(x) = \bar{s}'_i)$. It follows that $\bar{p}' \vdash \bar{s}'$.

What is left to show is $(\bar{v}, a) \in \mathcal{V}$ and $\forall i \in 1..n. (\bar{v}_i = \bullet \Rightarrow \bar{s}_i = \bar{s}'_i \wedge \bar{s}_i \in \mathcal{S}_i) \wedge (\bar{v}_i \neq \bullet \Rightarrow \bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i)$ (by Definition 4.2). Since $\mathcal{W} \subseteq \mathcal{V}$ (by APC3) and $(\bar{v}, a) \in \mathcal{W}$, we must have $(\bar{v}, a) \in \mathcal{V}$. Consider an $i \in 1..n$. We distinguish two cases:

- $\bar{v}_i = \bullet$. As $\bar{s} \in \mathcal{S}_{\mathcal{N}}$, it holds that $\bar{s}_i \in \mathcal{S}_i$. Moreover, by Definition 4.2, we have $\bar{p}_i = \bar{p}'_i$ and $\bar{p}_i \in \mathcal{S}_{\Phi_i^\kappa}$. If $\bar{p}_i = \kappa$, then by construction of \bar{s}' we have $\bar{s}_i = \bar{s}'_i$. If $\bar{p}_i \neq \kappa$, then $m_i(\bar{p}_i) = \bar{s}_i$ and $m_i(\bar{p}'_i) = \bar{s}'_i$. Finally, since $\bar{p}_i = \bar{p}'_i$ and m_i is injective (Definition 3.4), $m_i(\bar{p}_i) = m_i(\bar{p}'_i) = \bar{s}_i = \bar{s}'_i$.
- $\bar{v}_i \neq \bullet$. By Definition 4.2 and $(\bar{v}, a) \in \mathcal{W}$, we have $\bar{p}_i \xrightarrow{\bar{v}_i}_{\Phi_i} \bar{p}'_i$. Hence, it follows that $m_i(\bar{p}_i) = \bar{s}_i$ and $m_i(\bar{p}'_i) = \bar{s}'_i$. Finally, since a match (Definition 3.4) is an embedding, we conclude that $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$.

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

Lemma 4.7 Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ and a pattern network $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ of size n with $\mathcal{W} \subseteq \mathcal{V}$ (APC3) and corresponding matches $m_i : \bar{\Phi}_i \rightarrow \Pi_i$ for all $i \in 1..n$. Then,

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

Proof Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ and $\bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{N}}$ such that $\bar{p} \xrightarrow{\tau}_{\mathcal{P}^\kappa} \bar{p}'$, $\bar{p} \vdash \bar{s}$, and $\bar{p}' \vdash \bar{s}'$. Since τ result actions are not allowed in \mathcal{W}^κ , any law (\bar{v}, τ) in the sequence of τ 's must be a member of \mathcal{W} . The proof now follows directly from Lemma 4.6 and structural induction on $\bar{p} \xrightarrow{\tau}_{\mathcal{P}^\kappa} \bar{p}'$. \square

When two states \bar{s} and \bar{t} from the input and transformed network, respectively, are related by a branching bisimulation relation, and from \bar{s} a certain transition is enabled, then from \bar{t} it must be possible to simulate this behaviour (and vice versa). Even when this transition is not matched by the transformation rule system, it may still be the case that \bar{t} is matched by a state that is not a glue-state. In this case, \bar{t} cannot simulate \bar{s} directly, thus, there must be a τ -path to some other state that is able to directly simulate \bar{s} . The existence of such a state is proven in Lemma 4.8.

In Lemma 4.8, we show that in the presence of a state vector mapping and a witness showing that there is a transition leaving the pattern, all *active* glue-states (all glue-states involved in that transition) or κ -states are reachable by related states. More precisely, when a state \bar{p} – mapped to a state from which the pattern is left – is related to a state \bar{q} , then there is a τ -path from \bar{q} to \hat{q} such that $\bar{p} \leftrightarrow_b \hat{q}$ and there is a state \bar{q}' which is the corresponding entry-point of a given unmatched transition that leaves \bar{q} . This follows from two facts. First, the κ -synchronisation laws have synchronisation vectors uniquely identifying the active states within \bar{q} . Second, due to the matching conditions (Definition 3.4), a matched state must be a glue-state when there is a transition leaving or entering the corresponding matched state.

Lemma 4.8 Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system applicable on \mathcal{N} such that $\bar{\mathcal{L}}^\kappa \leftrightarrow_b \bar{\mathcal{R}}^\kappa$. Let \bar{M} be a vector of match pairs of size n such that $m_i : \mathcal{L}_i \rightarrow \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \rightarrow T(\Pi_i)$ for all $i \in 1..n$. Then,

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

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

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

We distinguish the two aforementioned cases:

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

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

Let μ be the unique result action corresponding to \bar{v}^κ . Since there are no matching transitions ($\forall i \in Ac(\bar{v})$. $\neg \bar{p}_i \xrightarrow{\bar{v}_i} \mathcal{L}_i \bar{p}'_i$), by Definition 3.4, for all $i \in Ac(\bar{v})$ we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. It follows that (\bar{v}^κ, μ) indeed enables the transition $\bar{p} \xrightarrow{\mu} \bar{\mathcal{L}}^\kappa \bar{p}'$. Furthermore, by Definition 4.6, we have $\mu \neq \tau$. Since $\bar{p} \xleftrightarrow{a} \bar{q}$ and $\mu \neq \tau$, by Definition 3.6, we have $\bar{q} \xrightarrow{\tau} \bar{\mathcal{R}}^\kappa \hat{q} \xrightarrow{\mu} \bar{\mathcal{R}}^\kappa \bar{q}'$ with $\bar{p} \xleftrightarrow{a} \hat{q}$ and $\bar{p}' \xleftrightarrow{a} \bar{q}'$. What remains to be shown is 1) $\forall i \in Ac(\bar{v})$. $\hat{q}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i$ and 2) $\forall i \in 1..n \setminus Ac(\bar{v})$. $\hat{q}_i = \bar{q}'_i$:

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

- $i \in Ac(\bar{v}^\kappa)$. Because μ is unique in \mathcal{V}^κ and does not occur in $\mathcal{V} \cup \hat{\mathcal{V}}$, the transition $\hat{q} \xrightarrow{\mu} \bar{\mathcal{R}}^\kappa \bar{q}'$ is enabled by $(\bar{v}^\kappa, \mu) \in \mathcal{V}^\kappa$. Recall that $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$ (since $i \in Ac(\bar{v})$). The \bar{v}_i^κ -transition is only present between \bar{p}_i and \bar{p}'_i in both $\bar{\mathcal{L}}^\kappa$ and $\bar{\mathcal{R}}^\kappa$, and (by Definition 3.7). Therefore, we must have $\hat{q}_i = \bar{p}_i$ and $\bar{q}'_i = \bar{p}'_i$.
- $i \notin Ac(\bar{v}^\kappa)$. It follows that $\bar{p}_i = \bar{p}'_i$ and $\hat{q}_i = \bar{q}'_i$ (Definition 4.2). Recall that $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$ (since $i \in Ac(\bar{v})$). However, since $i \notin Ac(\bar{v}^\kappa)$ only the case where both \bar{p}_i and \bar{p}'_i are κ -states remains. By applying Lemma 4.3 on $\bar{p} \xleftrightarrow{a} \hat{q}$ and $\bar{p}' \xleftrightarrow{a} \bar{q}'$, it follows that $\hat{q}_i = \kappa$ and $\bar{q}'_i = \kappa$. Hence, $\bar{p}_i = \bar{p}'_i$ and $\hat{q}_i = \bar{q}'_i$.

2) Consider an $i \in 1..n \setminus Ac(\bar{v})$. Since $i \notin Ac(\bar{v})$, we must have $i \notin Ac(\bar{v}^\kappa)$ (by construction of \bar{v}^κ). It follows from Definition 4.2 that $\hat{q}_i = \bar{q}'_i$.

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

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

We show that $\bar{p} = \bar{p}'$. Consider an $i \in 1..n$. If both $\bar{p}_i = \kappa$ and $\bar{p}'_i = \kappa$, we trivially have $\bar{p}_i = \bar{p}'_i$. Now consider the opposite case: $\bar{p}_i \neq \kappa$ or $\bar{p}'_i \neq \kappa$. Assume for a contradiction that $i \in Ac(\bar{v})$. Then by Definition 3.4, we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. Since $\bar{p}_i \neq \kappa$ or $\bar{p}'_i \neq \kappa$, only the following three cases remain: $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$, $(\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa)$, and $(\bar{p}_i = \kappa \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i})$. These three cases contradict the

assumptions of case C2. Hence, we must have $i \notin Ac(\bar{v})$. It now follows that $\bar{s}_i = \bar{s}'_i$. Finally, we have $\bar{p}_i = \bar{p}'_i$ because m is an injection. In conclusion, we have $\bar{p} = \bar{p}'$.

What remains to be shown is $\forall i \in Ac(\bar{v}). \bar{q}_i = \bar{q}'_i$. Consider an $i \in Ac(\bar{v})$. Again, by Definition 3.4, we must have $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \vee \bar{p}_i = \kappa$ and $\bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i} \vee \bar{p}'_i = \kappa$. Based on this we distinguish three cases: $\bar{p}_i = \kappa$, $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i = \kappa$, and $\bar{p}_i \in \mathcal{E}_{\bar{\mathcal{L}}_i} \wedge \bar{p}'_i \in \mathcal{I}_{\bar{\mathcal{L}}_i}$. In the first case it follows from $\bar{p} \leftrightarrow_b \bar{q}$ and Lemma 4.3 that $\bar{p}_i = \kappa = \bar{q}_i$. The latter two cases contradict one of the three assumptions of case C2 and the proof follows by contradiction.

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

Completeness of transition transformation. Due to the application conditions either all process-local transitions participating in a synchronising global transition are transformed or no process-local transitions are transformed at all. For confluent rule systems, the order of applying rules is irrelevant and application of the transformation rules can lead to only one output network. This fact allows us to simplify the correctness proof of the verification technique by strengthening condition **APC1** such that the correctness proof (Proposition 3.1) can be formulated on a single transformation step instead of a sequence.

Recall that **APC1** requires that a rule transforming synchronising transitions labeled with some action a must be applicable on all a -transitions within the corresponding LTS. Since confluent rule systems have only a single possible output network, we may consider a ‘merged’ match consisting of all the matches in the sequence that produces the output network in a single transformation step. A transformation sequence resulting in the final output network cannot be distinguished from the application of the ‘merged’ match. Therefore, for proving the correctness of the technique, **APC1** may be strengthened as follows:

$$\begin{aligned} \forall \Pi_i \in \Pi, r_j \in R, (\bar{v}, a) \in \mathcal{V}. \{j\} \subset Ac(\bar{v}) \wedge \bar{v}_j \in \mathcal{A}_{\mathcal{L}_j} \Rightarrow \\ \forall m: \mathcal{L}_j \rightarrow \Pi_i, (s, \bar{v}_j, s') \in \mathcal{I}_i. \exists (p, \bar{v}_j, p') \in \mathcal{I}_{\mathcal{L}_j}. m(p) = s \wedge m(p') = s' \end{aligned} \quad (\text{APC1}')$$

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

Conditions **APC1'** and **ANCI** ensure that global transitions in the input network are always fully transformed or not transformed at all. This leads to Lemma 4.9 that states the following: if a transition in a network enabled by $(\bar{v}, a) \in \mathcal{V}$ has a match on a local transition (say \bar{v}_i for some $i \in 1..n$), then for all $j \in 1..n$, the participating local \bar{v}_j -transitions must be matched, i.e., all local transition participating in the global transition must be matched. From this it follows that a global transition is either fully transformed or not transformed at all.

Lemma 4.9 Consider an LTS network $\mathcal{N} = (\Pi, \mathcal{V})$ of size n and a rule system $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ such that **APC1'** and **ANCI** are satisfied. Consider the pattern network Σ , let $\mathcal{P} = (\bar{\Phi}, \mathcal{W})$ as representative for the left and right pattern network. Let the $m_i : \bar{\Phi}_i \rightarrow \Pi_i$ ($i \in 1..n$) be the matches specifying the embedding of \mathcal{P} in \mathcal{N} . Then,

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

Proof Consider a synchronisation law $(\bar{v}, a) \in \mathcal{W}$ and states $\bar{s}, \bar{s}' \in \mathcal{S}_{\mathcal{N}}$ such that $\bar{s} \xrightarrow{\bar{v}, a}_{\mathcal{N}} \bar{s}'$. Let $\bar{p}, \bar{p}' \in \mathcal{S}_{\mathcal{P}^\kappa}$ with $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$. Finally, let there be an $j \in Ac(\bar{v})$ such that $m_j(\bar{p}_j) = \bar{s}_j$, $m_j(\bar{p}'_j) = \bar{s}'_j$, and $\bar{p}_j \xrightarrow{\bar{v}_j}_{\bar{\Phi}_j} \bar{p}'_j$ matches transition $\bar{s}_j \xrightarrow{\bar{v}_j}_j \bar{s}'_j$. We shall show that $\bar{p} \xrightarrow{\bar{v}, a}_{\mathcal{P}^\kappa} \bar{p}'$ by showing that for all $i \in 1..n$ there is a transition $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i} \bar{p}'_i$ if $\bar{v}_i \neq \bullet$, and $\bar{p}_i = \bar{p}'_i$ with $\bar{p}_i \in \mathcal{S}_{\bar{\Phi}_i}$ if $\bar{v}_i = \bullet$ (Definition 4.2). Consider an $i \in 1..n$. We distinguish three cases:

- $\bar{v}_i \neq \bullet \wedge Ac(\bar{v}) = \{i\}$. Law (\bar{v}, a) constitutes independent behaviour and the proof follows from the premises.

- $\bar{v}_i \neq \bullet \wedge \{i\} \subset Ac(\bar{v})$. Law (\bar{v}, a) constitutes synchronising behaviour. Next we show that $\bar{v}_i \in \mathcal{A}_{\bar{\Phi}_i}$, after which we can apply **APCI'** and show that $\bar{p}_i \xrightarrow{\bar{v}_i}_{\bar{\Phi}_i^\kappa} \bar{p}'_i$. Since **ANCI** is not symmetrical with respect $\bar{\mathcal{L}}$ and $\bar{\mathcal{R}}$ we need to distinguish these two cases showing $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$ and $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$ respectively.
 - $\mathcal{P} = \bar{\mathcal{L}}$. The left pattern network has the laws $\mathcal{W} = \mathcal{V}'$. Because $(\bar{v}, a) \in \mathcal{V}'$, by **ANCI**, we have $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$.
 - $\mathcal{P} = \bar{\mathcal{R}}$. The right pattern network has the laws $\mathcal{W} = \mathcal{V}' \cup \hat{\mathcal{V}}$, hence, $(\bar{v}, a) \in \mathcal{V}' \cup \hat{\mathcal{V}}$. In the trivial case, where $\bar{v} \in \hat{\mathcal{V}}$, it directly follows from **ANCI** that $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$. In the other case, where $\bar{v} \in \mathcal{V}'$, Since $i \in Ac(\bar{v})$, there is a transition $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$ in the transformed network. This transition either originates from the network Σ is applied on or is introduced by the transformation as specified in Definition 3.5. In the former case, we arrive at a contradiction: by the left variant of **APCI'** and **ANCI**, it follows that the original transition is matched on, while the transition is *not* matched on according to Definition 3.5. In the latter case, there exists $x, x' \in \mathcal{S}_{\mathcal{R}_i}$ with $x \xrightarrow{\bar{v}_i}_{\mathcal{R}_i} x'$. Therefore, we have $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i}$.

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

- $\bar{v}_i = \bullet$. By Definition 4.2, $\bar{s}_i = \bar{s}'_i$. Hence, since matches are injective it follows from $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$ that $\bar{p}_i = \bar{p}'_i$. Furthermore, since $\bar{p} \in \mathcal{S}_{\mathcal{P}^\kappa}$, we have $\bar{p}_i \in \mathcal{S}_{\bar{\Phi}_i^\kappa}$.

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

Soundness of the analysis. Proposition 4.1 formally describes the analysis technique. To show the soundness of Proposition 4.1, we have to prove that a branching bisimulation relation B between the κ -extended pattern networks of a transformation rule system implies, via state vector mappings, a branching bisimulation relation C between arbitrary original and transformed LTS networks.

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

A consequence of Lemma 4.9 is that two cases can be distinguished. If all process-local transitions are transformed, it follows that the state vector mapping preserves the branching structure of transitions enabled by non- κ -synchronisation laws. If no process-local transitions are transformed, it is still possible that a state \bar{s} is related via C to a state \bar{t} that is matched by at least one non-glue state; e.g., $\exists i \in 1..n, \bar{q}_i \in \mathcal{S}_{\mathcal{R}} \setminus \mathcal{S}_{\mathcal{L}}. \hat{m}_i(\bar{q}_i) = \bar{t}_i$. If \bar{s} is able to perform an a -transition enabled by a $(\bar{v}, a) \in \mathcal{V}$, then \bar{t} must be able to simulate this transition. Some local states \bar{t}_i ($i \in Ac(\bar{v})$) of \bar{t} may be matched on by non-glue states. In this case, \bar{t} is not able to perform the a -transition itself. Therefore, there must be a τ -path from \bar{t} to a state $\hat{\bar{t}}$ such that $\hat{\bar{t}}$ can perform this a -transition as is shown in Lemma 4.8.

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

Proposition 4.1 Let $\mathcal{N} = (\Pi, \mathcal{V})$ be an LTS network of size n and let $\Sigma = (R, \mathcal{V}', \hat{\mathcal{V}})$ be a rule system satisfying **ANCI**, **APCI'**, **APC2**, **APC3**, and **APC4**. Let \bar{M} be a vector of match pairs of size n such that $m_i : \mathcal{L}_i \rightarrow \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \rightarrow T(\Pi_i)$ for all $i \in 1..n$. Then,

$$(\forall P \in \mathbb{D}. \bar{\mathcal{L}}^{\kappa, P} \leftrightarrow_b \bar{\mathcal{R}}^{\kappa, P}) \Rightarrow \mathcal{N} \leftrightarrow_b T_{\bar{M}}(\mathcal{N})$$

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

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

To prove the proposition we have to show that C is a bisimulation relation. This requires proving that C relates the initial states of \mathcal{N} and $T_{\bar{M}}(\mathcal{N})$ and that C satisfies Definition 3.6.

- C relates the initial states of \mathcal{N} and $T_{\bar{M}}(\mathcal{N})$. We have $\mathcal{I}_{\mathcal{N}} = \mathcal{I}_{T_{\bar{M}}(\mathcal{N})}$. Hence, it suffices to show $\forall \bar{s} \in \mathcal{I}_{\mathcal{N}}. \bar{s} C \bar{s}$. Take an arbitrary state $\bar{s} \in \mathcal{I}_{\mathcal{N}}$, then by Lemma 4.5, there is a state $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^x}$ with $\bar{p} \vdash \bar{s}$. Since $\bar{s} \in \mathcal{I}_{\mathcal{N}}$, it follows from Definition 3.4 that $\forall i \in 1..n. \bar{p}_i \in \mathcal{E}_{\mathcal{L}_i} \vee \bar{p}_i = \kappa$, i.e., $\bar{p} \in \mathcal{I}_{\bar{\mathcal{L}}^x}$. By Lemma 4.4, we have $\bar{p} B \bar{p}$. It follows that $\bar{s} C \bar{s}$.
- If $\bar{s} C \bar{t}$ and $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$ then either $a = \tau \wedge \bar{s}' C \bar{t}$, or $\bar{t} \xrightarrow{\tau}_{T_{\bar{M}}(\mathcal{N})} \hat{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}' \wedge \bar{s} C \hat{t} \wedge \bar{s}' C \bar{t}'$. Consider synchronisation law $(\bar{v}, a) \in \mathcal{V}$ enabling the transition $\bar{s} \xrightarrow{a}_{\mathcal{N}} \bar{s}'$. Since $\bar{s} C \bar{t}$, there exist $\bar{p} \in \mathcal{S}_{\bar{\mathcal{L}}^x}$ and $\bar{q} \in \mathcal{S}_{\bar{\mathcal{R}}^x}$ such that $\bar{p} B \bar{q}$, $\bar{p} \vdash \bar{s}$, and $\bar{q} \vdash \bar{t}$, and $\forall i \in 1..n. (\bar{p}_i = \kappa \vee \bar{q}_i = \kappa) \Rightarrow \bar{s}_i = \bar{t}_i$ (2). Furthermore, by Lemma 4.5, there is a state $\bar{p}' \in \mathcal{S}_{\bar{\mathcal{L}}^x}$ with $\bar{p}' \vdash \bar{s}'$. We distinguish two cases:

1. $\exists i \in Ac(\bar{v}). m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^x} \bar{p}'_i$. We shall first establish that $(\bar{v}, a) \in \mathcal{V}'$, after which we can apply Lemma 4.9 to obtain the corresponding a -transition in $\mathcal{T}_{\bar{\mathcal{L}}^x}$. Assume for a contradiction that $(\bar{v}, a) \notin \mathcal{V}'$. Since there is an $i \in Ac(\bar{v})$ with $\bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^x} \bar{p}'_i$, the \bar{v}_i action must be a member of the actions of \mathcal{L}_i , i.e., $\bar{v}_i \in \mathcal{A}_{\mathcal{L}_i}$. However, since $i \in Ac(\bar{v})$ and $(\bar{v}, a) \in \mathcal{V} \setminus \mathcal{V}'$, by APC4, it must hold that $\bar{v}_i \notin \mathcal{A}_{\mathcal{L}_i}$. Hence, by contradiction, we have $(\bar{v}, a) \in \mathcal{V}'$.

Now, by Lemma 4.9, there is a transition $\bar{p} \xrightarrow{a}_{\bar{\mathcal{L}}^x} \bar{p}'$ enabled by (\bar{v}, a) . Since $\bar{p} B \bar{q}$, by Definition 3.6, we have the following two cases:

- $a = \tau$ and $\bar{p}' B \bar{q}$. To show $\bar{s} C \bar{t}$, all ingredients but one are there. In particular, we still need to show that $\forall i \in 1..n. \bar{p}'_i = \kappa \vee \bar{q}_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}_i = \kappa$. Since \mathcal{V}^x and \mathcal{V}' are disjoint, we must have $(\bar{v}, a) \notin \mathcal{V}^x$. If $\bar{q}_i = \kappa$, then by Lemma 4.3, $\bar{p}_i = \kappa$. Since $\bar{p}_i = \kappa$ or $\bar{p}'_i = \kappa$ and $(\bar{v}, a) \notin \mathcal{V}^x$, it follows that $i \notin Ac(\bar{v})$. Hence, by Definition 4.2, $\bar{s}'_i = \bar{s}_i$. Finally by (2), $\bar{s}_i = \bar{t}_i$.
- $\bar{q} \xrightarrow{\tau}_{\bar{\mathcal{R}}^x} \hat{q} \xrightarrow{a}_{\bar{\mathcal{R}}^x} \bar{q}'$ with $\bar{p} B \hat{q}$ and $\bar{p}' B \bar{q}'$. From Lemma 4.7, it follows that there is a state $\hat{t} \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\hat{q} \vdash \hat{t}$, a τ -path $\bar{t} \xrightarrow{\tau}_{T_{\bar{M}}(\mathcal{N})} \hat{t}$, and $\forall i \in 1..n. \hat{q}_i = \kappa \Rightarrow \hat{t}_i = \bar{t}_i$ (3). Furthermore, by Lemma 4.6, we have a state $\bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\bar{q}' \vdash \bar{t}'$, a transition $\hat{t} \xrightarrow{a}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, and $\forall i \in 1..n. \bar{q}'_i = \kappa \Rightarrow \bar{t}'_i = \hat{t}_i$ (4). What remains to be shown is that $\bar{s} C \hat{t}$ and $\bar{s}' C \bar{t}'$.
 - For $\bar{s} C \hat{t}$, all that is left to show is $\forall i \in 1..n. \bar{p}_i = \kappa \vee \hat{q}_i = \kappa \Rightarrow \bar{s}_i = \hat{t}_i$. Consider an $i \in 1..n$ with $\bar{p}_i = \kappa \vee \hat{q}_i = \kappa$. By Lemma 4.2, $\hat{q}_i = \kappa$ iff $\bar{q}_i = \kappa$. Hence, $\bar{p}_i = \kappa \vee \bar{q}_i = \kappa$. From (2), it follows that $\bar{s}_i = \bar{t}_i$. Finally, by Lemma 4.3, $\bar{p}_i = \kappa$ iff $\hat{q}_i = \kappa$ and it follows from (3) that $\hat{t}_i = \bar{t}_i$. Therefore, $\bar{s}_i = \bar{t}_i = \hat{t}_i$. In conclusion, $\bar{s} C \hat{t}$.
 - Similarly, for $\bar{s}' C \bar{t}'$, all that is left to show is $\forall i \in 1..n. \bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}'_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$. By Lemma 4.1, $\hat{q}_i = \kappa$ iff $\bar{q}'_i = \kappa$. Hence, $\bar{p}'_i = \kappa \vee \hat{q}_i = \kappa$. From $\bar{s} C \hat{t}$, it follows that $\bar{s}_i = \hat{t}_i$. Finally, by Lemma 4.3, $\bar{p}'_i = \kappa$ iff $\bar{q}'_i = \kappa$ and it follows from (4) that $\bar{t}'_i = \hat{t}_i$. Therefore, $\bar{s}'_i = \hat{t}_i = \bar{t}'_i$. In conclusion, $\bar{s}' C \bar{t}'$.

2. $\neg 1$. Because $\neg 1$, we have $\neg \exists i \in Ac(\bar{v}). m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^x} \bar{p}'_i$. That is, for all $i \in Ac(\bar{v})$ there is no transition in $\mathcal{T}_{\mathcal{L}_i}$ matching on $\bar{s}_i \xrightarrow{\bar{v}_i} \bar{s}'_i$, or more formally, $\neg(m_i(\bar{p}_i) = \bar{s}_i \wedge m_i(\bar{p}'_i) = \bar{s}'_i \wedge \bar{p}_i \xrightarrow{\bar{v}_i}_{\mathcal{L}_i^x} \bar{p}'_i)$. Therefore, for all states $\bar{p}, \bar{p}' \in \mathcal{S}_{\bar{\mathcal{L}}^x}$ with $\bar{p} \vdash \bar{s}$ and $\bar{p}' \vdash \bar{s}'$, there is no transition $\bar{p} \xrightarrow{a}_{\bar{\mathcal{L}}^x} \bar{p}'$. Moreover, by Definitions 3.7 and 3.4, for each $i \in 1..n$ state \bar{p}_i is either a κ -state or an exit-state (in $\mathcal{E}_{\mathcal{L}_i}$), and state \bar{p}'_i is either a κ -state or an in-state (in $\mathcal{I}_{\mathcal{L}_i}$), i.e., $\forall i \in Ac(\bar{v}). (\bar{p}_i \in \mathcal{I}_{\mathcal{L}_i} \vee \bar{p}_i \in \mathcal{E}_{\mathcal{L}_i}) \wedge (\bar{p}'_i \in \mathcal{I}_{\mathcal{L}_i} \vee \bar{p}'_i = \kappa)$ (5). By applying Lemma 4.8, we get states $\hat{q}, \bar{q}' \in \mathcal{S}_{\bar{\mathcal{R}}^x}$ such that there is a τ -path $\bar{q} \xrightarrow{\tau}_{\bar{\mathcal{R}}^x} \hat{q}$ with related states $\bar{p} B \hat{q}$ and $\bar{p}' B \bar{q}'$, and the states have the following two properties: $\forall i \in Ac(\bar{v}). \hat{q}_i = \bar{p}_i \wedge \bar{q}'_i = \bar{p}'_i$ (6), and $\forall i \in 1..n \setminus Ac(\bar{v}). \hat{q}_i = \bar{q}'_i$ (7).

From Lemma 4.7 it follows that there is a state $\hat{t} \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ with $\hat{q} \vdash \hat{t}$, a τ -path $\bar{t} \xrightarrow{\tau}_{T_{\bar{M}}(\mathcal{N})} \hat{t}$, and $\forall i \in 1..n. \hat{q}_i = \kappa \Rightarrow \hat{t}_i = \bar{t}_i$ (8). We construct a state $\bar{t}' := \hat{t}[\bar{s}'_i \mid i \in Ac(\bar{v})]$. By construction of \bar{t}' we

have $\forall i \in Ac(\bar{v}). \bar{t}'_i = \bar{s}'_i$ and $\forall i \in 1..n \setminus Ac(\bar{v}). \bar{t}'_i = \hat{t}_i$. To prove that $\hat{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, what remains to be shown is $\forall i \in Ac(\bar{v}). \hat{t}_i = \bar{s}_i$: consider an $i \in Ac(\bar{v})$. By (6), $\hat{q}_i = \bar{p}_i$. If $\hat{q}_i = \kappa$, then also $\bar{p}_i = \kappa$ (Lemma 4.3). Therefore, by (8) and (2), $\hat{t}_i = \bar{t}_i = \bar{s}_i$. If $\hat{q}_i \neq \kappa$, then also $\bar{p}_i \neq \kappa$ (Lemma 4.3). It follows that $m_i(\bar{p}_i) = \bar{s}_i$ and $\hat{m}_i(\hat{q}_i) = \hat{t}_i$. By (5) and $\bar{p}_i \neq \kappa$, it holds that $\bar{p}_i \in \mathcal{E}_{\mathcal{L}_i}$. Thus, $\hat{m}_i(\bar{p}_i) = \bar{s}_i$ (by Definition 3.5). Finally, by injectivity of \hat{m} we have $\hat{t}_i = \bar{s}_i$.

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

- For $\bar{q}' \vdash \bar{t}'$, we have to show that $\forall i \in 1..n. \bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$ (i.e., $\bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$) and $\forall i \in 1..n. (\bar{q}'_i \neq \kappa \Rightarrow \hat{m}_i(\bar{q}'_i) = \bar{t}'_i) \wedge (\bar{q}'_i = \kappa \Rightarrow \neg \exists x \in \mathcal{S}_{\mathcal{R}_i}. \hat{m}_i(x) = \bar{t}'_i)$. Consider an $i \in 1..n$. Based on the construction of \bar{t}' we distinguish the following cases:
 - $i \in Ac(\bar{v})$. We have $\bar{t}'_i = \bar{s}'_i$ (by construction of \bar{t}') and $\bar{q}'_i = \bar{p}'_i$ (by (6)). By (5) and $\bar{q}'_i = \bar{p}'_i$, either $\bar{q}'_i \in \mathcal{I}_{\mathcal{L}_i}$ or $\bar{q}'_i = \kappa$.
If $\bar{q}'_i \in \mathcal{I}_{\mathcal{L}_i}$, then also $\bar{p}'_i \in \mathcal{I}_{\mathcal{L}_i}$ and we have $m_i(\bar{p}'_i) = \bar{s}'_i$ which we may rewrite to $\hat{m}_i(\bar{p}'_i) = \bar{s}'_i$ (by Definition 3.5). Finally, by $\bar{q}'_i = \bar{p}'_i$ and $\bar{t}'_i = \bar{s}'_i$, we have $\hat{m}_i(\bar{q}'_i) = \bar{s}'_i = \bar{t}'_i$. Furthermore, since $\hat{m}_i(\bar{q}'_i) = \bar{t}'_i$, it follows that $\bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$.
If $\bar{q}'_i = \kappa$, then we have to show $\neg \exists x \in \mathcal{S}_{\mathcal{R}_i}. \hat{m}_i(x) = \bar{t}'_i$. Assume for a contradiction that there is a state $x \in \mathcal{S}_{\mathcal{R}_i}$ such that $\hat{m}_i(x) = \bar{t}'_i$. Since $\bar{s}_i \xrightarrow{\bar{v}_i}_i \bar{s}'_i$ is not matched on, by Definition 3.4, we must have $x \in \mathcal{I}_{\mathcal{L}_i}$. By $x \in \mathcal{I}_{\mathcal{L}_i}$ and $\bar{t}'_i = \bar{s}'_i$ (by construction of \bar{t}'), we have $m_i(x) = \hat{t}_i$. However, since $\bar{q}'_i = \bar{p}'_i$ (by (6)), we have $\bar{p}'_i = \kappa$. Thus, by Definition 4.8, there is no such x with $m_i(x) = \hat{t}_i$. Furthermore, since \bar{s}' is not matched on by m_i , the state remains present in $T(\Pi_i)$. Hence, since $\bar{t}'_i = \bar{s}'_i$, it holds that $\bar{t}'_i \in \mathcal{S}_{T(\Pi_i)}$.
 - $i \notin Ac(\bar{v})$. We have $\bar{t}'_i = \hat{t}_i$ (by construction of \bar{t}') and $\bar{q}'_i = \hat{q}_i$ (by (7)). The proof now follows directly by substituting \bar{t}'_i for \hat{t}_i and substituting \hat{q}_i for \bar{q}'_i in $\hat{q} \vdash \hat{t}$.
- For $\bar{s} \hat{C} \hat{t}$, we still have to show that $\forall i \in 1..n. \bar{p}_i = \kappa \vee \hat{q}_i = \kappa \Rightarrow \bar{s}_i = \hat{t}_i$. Consider an $i \in 1..n$ with $\bar{p}_i = \kappa \vee \hat{q}_i = \kappa$. By Lemma 4.3, $\hat{q}_i = \kappa$ iff $\bar{p}_i = \kappa$. Hence, by (8), we have $\hat{t}_i = \bar{t}_i$. Finally, by (2) $\bar{s}_i = \bar{t}_i$, thus we have $\hat{t}_i = \bar{t}_i = \bar{s}_i$. In conclusion, $\bar{s} \hat{C} \hat{t}$.
- Similarly, for $\bar{s}' \hat{C} \bar{t}'$, all that is left to show is that $\forall i \in 1..n. \bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa \Rightarrow \bar{s}'_i = \bar{t}'_i$. Consider an $i \in 1..n$ with $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$. If $i \in Ac(\bar{v})$, then by construction of \bar{t}' , we have $\bar{s}'_i = \bar{t}'_i$. Conversely, if $i \notin Ac(\bar{v})$, then $\bar{t}'_i = \hat{t}_i$ and $\bar{s}'_i = \bar{s}_i$. Hence, also $\bar{q}'_i = \hat{q}_i$ and $\bar{p}'_i = \bar{p}_i$. Since $\bar{p}'_i = \kappa \vee \bar{q}'_i = \kappa$, it now follows that $\bar{p}_i = \kappa \vee \hat{q}_i = \kappa$. By $\bar{s} \hat{C} \hat{t}$, we have $\bar{s}_i = \hat{t}_i$, thus $\bar{s}'_i = \bar{s}_i = \hat{t}_i = \bar{t}'_i$. In conclusion, $\bar{s}' \hat{C} \bar{t}'$.
- *The symmetric case: if $\bar{s} \hat{C} \bar{t}$ and $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ then either $a = \tau \wedge \bar{s} \hat{C} \bar{t}'$, or $\bar{s} \xrightarrow{\tau}_{\mathcal{N}} \hat{s} \xrightarrow{\alpha}_{\mathcal{N}} \bar{s}' \wedge \hat{s} \hat{C} \bar{t} \wedge \bar{s}' \hat{C} \bar{t}'$. This case is symmetric to the previous case, with the exception that $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is enabled by some $(\bar{v}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. Therefore, when transition $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is not matched on, we have to show that $(\bar{v}, a) \in \mathcal{V}$. Let $\bar{t}, \bar{t}' \in \mathcal{S}_{T_{\bar{M}}(\mathcal{N})}$ such that $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is enabled by some $(\bar{v}, a) \in \mathcal{V} \cup \hat{\mathcal{V}}$. Furthermore, transition $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$ is not matched on. Assume for a contradiction that $(\bar{v}, a) \in \hat{\mathcal{V}}$. Since $(\bar{v}, a) \in \hat{\mathcal{V}}$ is introduced by the transformation, by APC2, for all $i \in Ac(\bar{v})$ the action \bar{v}_i does not occur in the original process Π_i , i.e., for all $i \in 1..n$, we have $\bar{v}_i \notin \mathcal{A}_i$. Hence, these actions \bar{v}_i must be introduced by \mathcal{R}_i , i.e., $\bar{v}_i \in \mathcal{A}_{\mathcal{R}_i} \setminus \mathcal{A}_{\mathcal{L}_i}$. It follows that there is a transition matching $\bar{t} \xrightarrow{\alpha}_{T_{\bar{M}}(\mathcal{N})} \bar{t}'$, contradicting our earlier assumption. Hence, we must have $(\bar{v}, a) \in \mathcal{V}$.*

□

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

Similarly to the analysis of a single transformation rule this analysis considers all input LTS networks that satisfy the analysis and application conditions. Hence, even when a rule system does *not* preserve a given property it may still be the case that the property is preserved for some instances of the transformation. For instance, given

a rule system Σ that is not property preserving there may be an input network \mathcal{N}' with a vector of matches \bar{M}' such that $\mathcal{N}' \xleftrightarrow{b} T_{\Sigma}(\mathcal{N}')$. However, it is guaranteed for Σ that there also exists an LTS network \mathcal{N}'' and vector of matches \bar{M}'' such that $\mathcal{N}'' \not\xleftrightarrow{b} T_{\Sigma}(\mathcal{N}'')$.

Proposition 4.2 Consider a rule system $\Sigma = (R, \mathcal{V})$. Let \mathbb{M} be the set of all LTS networks and $\Sigma_{\mathcal{N}}$ be the set of all possible vectors \bar{M} of n match pairs defining a transformation step using Σ for an LTS network $\mathcal{N} = (\Pi, \mathcal{V}) \in \mathbb{M}$ of size n . Such a vector \bar{M} consists of tuples (m_i, \hat{m}_i) , with $m_i : \mathcal{L}_i \rightarrow \Pi_i$ and $\hat{m}_i : \mathcal{R}_i \rightarrow T(\Pi_i)$, respectively. The following holds:

$$(\forall \mathcal{N} \in \mathbb{M}, \bar{M} \in \Sigma_{\mathcal{N}}. \mathcal{N} \xleftrightarrow{b} T_{\bar{M}}(\mathcal{N})) \Rightarrow \bar{\mathcal{L}}^{\kappa} \xleftrightarrow{b} \bar{\mathcal{R}}^{\kappa}$$

Proof Assume that for all $\mathcal{N} \in \mathbb{M}$ and $\bar{M} \in \Sigma_{\mathcal{N}}$ it holds that $\mathcal{N} \xleftrightarrow{b} T_{\bar{M}}(\mathcal{N})$. Trivially, we have $\bar{\mathcal{L}}^{\kappa} \in \mathbb{M}$ and trivial matches $(m_i : \bar{\mathcal{L}}_i \rightarrow \bar{\mathcal{L}}_i^{\kappa}, \hat{m}_i : \bar{\mathcal{R}}_i \rightarrow T(\bar{\mathcal{L}}_i^{\kappa}))$ (for each $i \in 1..n$) constituting a vector of matches \bar{M} . It follows from the assumption that $\bar{\mathcal{L}}^{\kappa} \xleftrightarrow{b} T_{\Sigma}(\bar{\mathcal{L}}^{\kappa})$. By Definition 4.6, $\bar{\mathcal{L}}^{\kappa} = (\langle \mathcal{L}_1^{\kappa}, \dots, \mathcal{L}_n^{\kappa} \rangle, \mathcal{V}' \cup \mathcal{V}^{\kappa})$ and $\bar{\mathcal{R}}^{\kappa} = (\langle \mathcal{R}_1^{\kappa}, \dots, \mathcal{R}_n^{\kappa} \rangle, \mathcal{V}' \cup \hat{\mathcal{V}} \cup \mathcal{V}^{\kappa} \cup \hat{\mathcal{V}}^{\kappa})$. By Definition 4.5, we have $T_{\Sigma}(\bar{\mathcal{L}}^{\kappa}) = (\langle \mathcal{R}_1^{\kappa} \dots \mathcal{R}_n^{\kappa} \rangle, \mathcal{V}' \cup \mathcal{V}^{\kappa} \cup \hat{\mathcal{V}} \cup \hat{\mathcal{V}}^{\kappa}) = \bar{\mathcal{R}}^{\kappa}$. It follows that $\bar{\mathcal{L}}^{\kappa} \xleftrightarrow{b} \bar{\mathcal{R}}^{\kappa}$. \square

5. Experiments

REFINER is implemented in Python 3 and can be run from the command-line. It is platform-independent, and allows performing behavioural transformations of LTS networks, and checking semantics and property preservation. It integrates with the action-based, explicit-state model checking toolsets CADP [GLMS11] and MCRL2 [CGK⁺13]. These tools can be used to specify and verify concurrent systems. REFINER uses the MCRL2 tool LTSCOMPARE to perform bisimilarity comparisons with an implementation of the Groote-Wijs algorithm [GW16].

For the experiments presented in this section REFINER was compiled using Nuitka to get a free performance improvement.⁶ We ran REFINER on the standard machines of the DAS-5 cluster [BE⁺DL⁺16], which have an INTEL HASWELL E5-2630-v3 2.4 GHz processor, 64 GB memory, running CENTOS LINUX 7.2. Each experiment was conducted no longer than 80 hours and aborted in case the machine ran out of memory.

We have performed two types of experiments. The *first* setup compares traditional model checking with the transformation verification approach presented in this work. The results are reported in Sect. 5.1. The *second* setup aims to compare the previous transformation verification algorithm (REFINER v1) with the algorithm in presented in this paper (REFINER v2). Those results are discussed in Sect. 5.2.⁷

5.1. Comparing traditional model checking and transformation verification

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

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

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

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

For CADP only the execution time of the tool can be measured. Hence, to make a fair comparison, we measured the execution time of the REFINER v1 tool in the same way. For both tools we have measured the wall clock time (i.e., the real elapsed time) using the Unix time command:

⁶ <http://nuitka.net>

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

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

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

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

The command used to verify a network using CADP is:

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

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

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

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

The argument `-q` indicates that REFINER should run in quiet mode, i.e., no messages are sent to the standard output. The `-c2` argument tells REFINER to use the verification algorithm presented in this article. Next, `-c <rule_system>` indicates that REFINER will verify the rule system `<rule_system>`. Finally, `-p <property>` specifies the property that REFINER verifies to be preserved, and `-f` indicates that τ -loops should be ignored. Without the latter argument, REFINER checks for branching bisimilarity *with explicit divergence* [vGW96, WE13] of the rule networks to determine whether liveness properties are preserved. For safety properties and inevitable reachability properties, this check is not required, and instead, branching bisimulation checking suffices.

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

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

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

The set of test cases. As test input, we selected nine case studies, two newly created ones, three from the set of MCRL2 models distributed with its toolset, and four from the set of CADP models.

The newly created ones are the following:

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

The models stemming from the MCRL2 toolset distribution are the following:

1. The 1394-fin model describes the 1394 or firewire protocol. It has been created by Luttk [Lut97].
2. The ACS model describes a part of the software of the Alma project of the European Southern Observatory, which involves controlling a large collection of radio telescopes. It consists of a manager and some containers and components. The model was created by Ploeger [Plo09].
3. Wafer stepper is a model of a wafer stepper.

Finally, the CADP models are the following:

1. ODP is a model of an open distributed processing trader [GS99].
2. The DES model describes an implementation of the data encryption standard, which allows to cipher and decipher 64-bit vectors using a 64-bit key vector [Nat99].
3. HAVi-LE describes the asynchronous Leader Election protocol used in the HAVi (Home Audio-Video) standard, involving three device control managers. The model is fully described by Romijn [Rom99].
4. Erat. Sieve is a specification of a distributed Eratosthenes sieve. It consists of a number generator and a chain of 17 units, each unit i filtering out the i^{th} prime number.

Table 1. Experimental results: verification of various models using CADP and REFINER

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

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

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

In the *second* and *third* columns metrics obtained from verifying the test model using CADP are displayed. We report the number of states each state space consists of (*second* column), and the running time (in seconds) to generate and verify these using CADP (*third* column).

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

Finally, the *fifth* column provides the outcome of the verification for each case, where ✓ indicates that the system satisfies the property and ✗ indicates that it does not.

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

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

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

5.2. Refiner v1 Versus Refiner v2

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

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

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

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

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

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

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

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

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

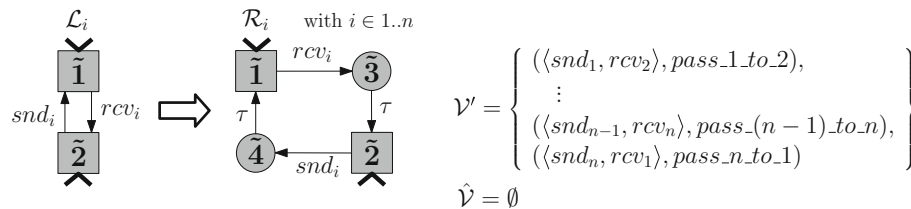
For REFINER v1 using multiple threads we used the command:

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

Without the `-t` argument REFINER creates a thread for each core of the machine and distributes the checks over these threads. In the case of a standard DAS-5 machine eight threads are used. The remaining arguments are the same as the ones for the single threaded variant.

REFINER v2 is invoked using the following command:

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

Fig. 13. Rule system transforming a token ring of size n Table 2. Experimental results: verification of token ring rule systems of size n using REFINER v1 and REFINER v2

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

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

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

The rule systems considered for this experiment model the transformation of token rings of size n . The network topology of a token ring ensures that the rule system consists of one dependency set. A generic representation of these rule systems is shown in Fig. 13. For a generated rule system of size n there are n transformation rules and n synchronisation laws. The action-labels snd_i and rcv_i indicate that the i^{th} rule or node performs a *send* and *receive* action, respectively.

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

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

Discussion of results. The results of this experiment are presented in Table 2. The size n of the rule system model is indicated by the *first* column. Each row shows the results of the verification for the rule system model of size n . The *second* and *third* columns describe the size of the right κ -extended pattern network in terms of number of states and transitions, respectively. The right κ -extended pattern network is the larger of the two networks, therefore, it gives a good indication of the size of the state space. The *fourth*, *fifth*, and *sixth* columns present the averaged running time per model in seconds for single-threaded REFINER v1, multi-threaded REFINER v1, and REFINER v2, respectively. Each experiment was conducted ten times.

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

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

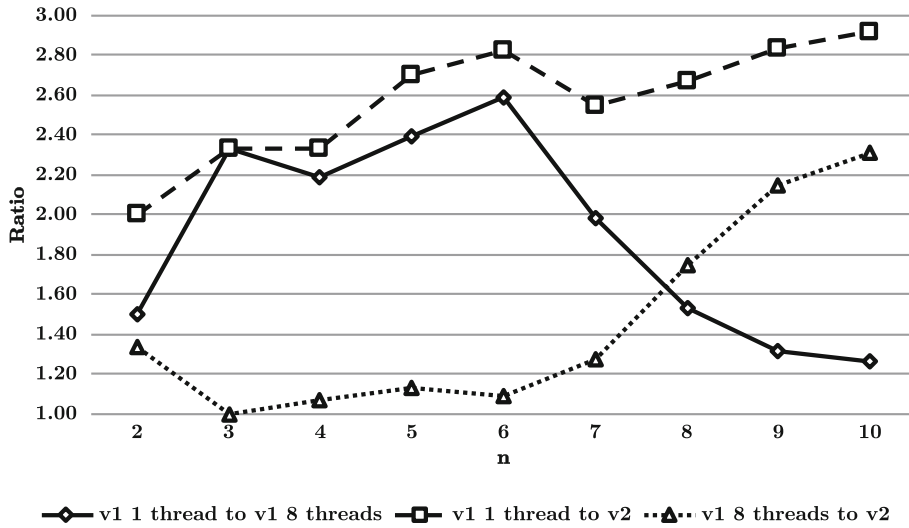


Fig. 14. Ratios between the REFINER versions when analysing the transformation of a token ring of size n

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

A last observation we can make based on Table 2 is that the running time ratio of REFINER v1 to v2 seems to increase. To investigate whether this is a trend we have plotted the ratios between the different REFINER versions in Fig. 14. The *horizontal axis* depicts the number of rules in the generated rule system, the *vertical axis* indicates the ratio. Although the number of data points is limited, the graph gives us some insights into the practical running time improvements.

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

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

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

The *dotted line* shows the ratio of REFINER v1 running 8 threads to REFINER v2 where the data points are indicated with triangles. This ratio shows an increase as n grows similar to the ratio between the single threaded REFINER v1 and REFINER v2. As n grows the data points move towards the dashed line (the single threaded REFINER v1 to REFINER v2 ratio). This is expected as the difference between the single threaded REFINER v1 and multi-threaded REFINER v1 decreases as n grows as indicated by the continuous line. At three rules, there are exactly

as many cores as there are checks for REFINER v1. Hence, at this point the performance of the multi-threaded REFINER v1 is equivalent to that of REFINER v2. However, at two rules, there are more cores than checks, but REFINER v2 performs better than REFINER v1. As the checks are extremely small for rule systems consisting of 2 rules it is likely that the overhead of the threads have a visible impact on the performance of REFINER v1.

6. Conclusions

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

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

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

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

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

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

Future work. In earlier work, we used branching bisimulation *with explicit divergence* [vGW96, WE13], which preserves τ -loops and therefore liveness properties. In future work, we would like to prove that for this flavour of bisimulation the technique is also correct. Moreover, we would like to investigate what the practical limitations of the pre-conditions of the technique are in industrial sized transformation systems.

In [Wij13], the technique from [WE13] has been extended to explicitly consider the communication interfaces between components, thereby removing the completeness condition ANC1 regarding synchronising behaviour being transformed (see Sect. 4.1). We wish to prove that also this extension is correct.

Finally, our framework can be extended in a number of ways, to reason about additional aspects of concurrent systems. For instance, in line with the encoding proposed in [Wij07], timing information could be included in the LTSs to design timed systems and express transformations of timed behaviour. This would also introduce the possibility to analyse the impact a transformation will have on the performance of a system under transformation [WF05], by means of timed branching bisimulation checking [FPW05]. The capability to reason about system performance could be further strengthened by also introducing probabilities on the LTS transitions [BK08]. Existing tools, such as PRISM [KNP11] and extensions [BESW10], could then be employed to conduct the analysis of the systems. An interesting challenge is then how to involve these probabilities in the verification of transformations as well.

Acknowledgements

This work is supported by ARTEMIS/ECSEL Joint Undertaking project EMC2 (grant nr. 621429). We would like to thank our anonymous reviewers for the insights and feedback that helped improve the quality of this work. Finally, we would like to thank Mark van den Brand for our fruitful discussions.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- [ABH⁺10] Abrial J-R, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) RODIN: an open toolset for modelling and reasoning in EVENT-B. *Softw Tools Technol Transf* 12(6):447–466
- [ACL⁺15] Amrani M, Combemale B, Lúcio L, Selim GMK, Dingel J, Le Traon Y, Vangheluwe H, Cordy JR (2015) Formal verification techniques for model transformations: a tridimensional classification. *J Object Technol* 14(3):1–43
- [AL91] Abadi M, Lamport L (1991) The existence of refinement mappings. *Theor Comput Sci* 82:253–284
- [BCE⁺07] Baldan P, Corradini A, Ehrig H, Heckel R, König B (2007) Bisimilarity and behaviour-preserving reconstructions of open petri nets. In: *Proceeding of 2nd conference on algebra and coalgebra in computer science (CALCO 2007)*, volume 4624 of LNCS. Springer, pp 126–142
- [BE04] Braunstein C, Encrenaz E (2004) CTL-property transformation along an incremental design process. In: *Proceeding of 4th international workshop on automated verification of critical systems*, volume 128 of ENTCS. Elsevier, pp 263–278
- [BE⁺L16] Bal H, Epema D, de Laat C, van Nieuwpoort R, Romein J, Seinstra F, Snoek C, Wijshoff H (2016) A medium-scale distributed system for computer science research: infrastructure for the long term. *IEEE Comput* 49(5):54–63
- [BESW10] Bošnački D, Edelkamp S, Sulewski D, Wijs AJ (2010) GPU-PRISM: an extension of PRISM for general purpose graphics processing units. In: *Proceeding of 9th international workshop on parallel and distributed methods in verification (PDMC 2010)*. IEEE Computer Society Press, pp 17–19
- [BGL05] Blech JO, Glesner S, Leitner J (2005) Formal verification of java code generation from UML models. In: *3rd international fujaba days*. Fujaba Days, pp 49–56
- [BH04] Bowen JP, Hinchey MG (2004) Formal methods. In: *Computldbook*, chapter 106. ACM, pp 106–1–106–25
- [BK08] Baier C, Katoen J-P (2008) Principles of model checking. MIT Press, Cambridge
- [CCGT09] Combemale B, Crégut X, Garoche P-L, Thirioux X (2009) Essay on semantics definition in MDE: an instrumented approach for model verification. *J Softw* 4(9):943–958
- [CGK⁺13] Cranen S, Grootte JF, Keiren JJA, Stappers FPM, de Vink EP, Wesselink W, Willemse T (2013) An overview of the mCRL2 toolset and its recent advances. In: *Proceeding of 19th international conference on tools and algorithms for the construction and analysis of systems (TACAS 2013)*, volume 7795 of LNCS. Springer, pp 199–213
- [DP06] Dodds M, Plump D (2006) Graph transformation in constant time. In: *Proceeding of 3rd international conference on graph transformation (ICGT 2006)*, volume 4178 of LNCS. Springer, pp 367–382
- [EGI97] Eppstein D, Galil Z, Italiano G (1997) Dynamic graph algorithms. In: *CRC handbook of algorithms and theory of computation* chapter 22. CRC Press, Boca Raton
- [FPW05] Fokkink WJ, Pang J, Wijs AJ (2005) Is timed branching bisimilarity an equivalence indeed? In: *Proceeding of 3rd conference on formal modelling and analysis of timed systems (FORMATS 2005)*, volume 3829 of Lecture Notes in Computer Science. Springer, pp 258–272
- [GGL⁺06] Giese H, Glesner S, Leitner J, Schäfer W, Wagner R (2006) Towards verified model transformations. In: *Proceeding of 3rd workshop on model-driven engineering, verification, and validation (MoDeVvA 2006)*, pp 78–93
- [GL12] Giese H, Lambers L (2012) Towards automatic verification of behavior preservation for model transformation via invariant checking. In: *Proceeding of 6th international conference on graph transformation (ICGT 2012)*, volume 7562 of LNCS. Springer, pp 249–263
- [GLMS11] Garavel H, Lang F, Mateescu R, Serwe W (2011) CADP 2010: A toolbox for the construction and analysis of distributed processes. In: *Proceeding of 17th international conference on tools and algorithms for the construction and analysis of systems (TACAS 2011)*, volume 6605 of LNCS. Springer, pp 372–387
- [GS99] Garavel H, Sighireanu M (1999) A graphical parallel composition operator for process algebras. In: *Proceeding of 1999 IFIP TC6/WG6.1 joint international conference on formal description techniques and protocol specification, testing and verification (FORTE/PSTV 1999)*, volume 156 of IFIP conference proceedings. Kluwer, pp 185–202
- [GW16] Grootte JF, Wijs AJ (2016) An $O(m \log n)$ algorithm for stuttering equivalence and branching bisimulation. In: *Proceeding of 22nd international conference on tools and algorithms for the construction and analysis of systems (TACAS 2016)*, volume 9636 of LNCS. Springer, pp 607–624
- [HKR⁺10] Hülsbusch M, König B, Rensink A, Semenyak M, Soltenborn C, Wehrheim H (2010) Showing full semantics preservation in model transformations: a comparison of techniques. In: *Proceeding of 8th international conference on integrated formal methods (iFM 2010)*, volume 6396 of LNCS, pp 183–198. Springer.
- [KLG07] Kundu S, Lerner S, Gupta R (2007) Automated refinement checking of concurrent systems. In: *Proceeding of 26th international conference on computer-aided design (ICCAD 2007)*. IEEE, pp 318–325
- [KN07] Karsai G, Narayanan A (2007) On the correctness of model transformations in the development of embedded systems. In: *Proceeding of 13th monterey workshop 2006*, volume 4888 of LNCS. Springer, pp 1–18
- [KNP11] Kwiatkowska M, Norman G, Parker D (2011) PRISM 4.0: verification of probabilistic real-time systems. In: *Proceeding of 23rd international conference on computer aided verification (CAV 2011)*, volume 6806 of LNCS. Springer, pp 585–591
- [KR08] Kahsay T, Roggenbach M (2008) Property preserving refinement for CSP- CASL. In: *Proceeding of 19th international workshop on algebraic development techniques (WADT 2008)*, volume 5486 of LNCS. Springer, pp 206–220
- [KWB05] Kleppe A, Warmer J, Bast W (2005) MDA Explained: The Model Driven Architecture(TM): Practice and Promise. Addison-Wesley Professional
- [Lan96] Lano K (1996) The B language and method, a guide to practical formal development. Springer, New York
- [Lan06] Lang F (2006) Refined interfaces for compositional verification. In: *Proceeding of 26th international conference on formal methods for networked and distributed systems (FORTE 2006)*, volume 4229 of LNCS. Springer, pp 159–174
- [LM13] Lang F, Mateescu R (2013) Partial model checking using networks of labelled transition systems and boolean equation systems. *Log Methods Comput Sci* 9(4):1–32.
- [Lut97] Lutтик SP (1997) Description and formal specification of the link layer of P1394. Technical Report SEN-R9706, CWI

- [MW14] Mateescu R, Wijs AJ (2014) Property-dependent reductions adequate with divergence-sensitive branching bisimilarity. *Sci Comput Program* 96(3):354–376.
- [Nat99] National Institute of Standards and Technology (1999) Data encryption standard (DES). Federal information processing standards pp 46–3
- [NK08] Narayanan A, Karsai G (2008) Towards verifying model transformations. In: *Proceeding of 7th international workshop on graph transformation and visual modeling techniques (GT-VMT 2008)*, volume 211 of ENTCS. Elsevier, pp 191–200
- [Plo09] Ploeger B (2009) Analysis of ACS using mCRL2. Technical Report 09-11, Eindhoven University of Technology
- [PW16] de Putter S, Wijs AJ (2016) Verifying a verifier: on the formal correctness of an LTS transformation verification technique. In: *Proceeding of 19th international conference on fundamental approaches to software engineering (FASE 2016)*, volume 9633 of LNCS. Springer, pp 383–400
- [Rom99] Romijn J (1999) Model checking a HAVi leader election protocol. Technical Report SEN-R9915, CWI
- [RR96] Ramalingam G, Reps T (1996) On the computational complexity of dynamic graph problems. *Theor Comput Sci* 158:233–277.
- [RW13] Rahim LA, Whittle J (2013) A survey of approaches for verifying model transformations. *Softw Syst Model* pp 1–26.
- [Sah07] Saha D (2007) An incremental bisimulation algorithm. In: *Proceeding of 27th iarc annual conference on foundations of software technology and theoretical computer science (FSTTCS 2007)*, volume 4855 of LNCS. Springer, pp 204–215
- [SLC⁺14] Selim GMK, Lúcio L, Cordy JR, Dingel J, Oakes BJ (2014) Specification and verification of graph-based model transformation properties. In: *Proceeding of 9th international colloquium on graph theory and combinatorics (ICGT 2014)*, volume 8571 of LNCS. Springer, pp 113–129
- [SMR11] Stenzel K, Moebius N, Reif W (2011) Formal verification of QVT transformations for code generation. In: *Proceeding of 14th international conference on model driven engineering languages and systems (MODELS 2011)*, volume 6981 of LNCS. Springer, pp 533–547
- [SS94] Sokolsky OV, Smolka SA (1994) Incremental model checking in the modal μ -Calculus. In: *Proceeding of 6th international conference on computer aided verification (CAV 1994)*, volume 818 of LNCS. Springer, pp 351–363
- [Swa96] Swamy GM (1996) Incremental methods for formal verification and logic synthesis. PhD thesis, University of California
- [vGW96] van Glabbeek RJ, Weijland WP (1996) Branching time and abstraction in bisimulation semantics. *J ACM* 43(3):555–600
- [VP03] Varró D, Pataricza A (2003) Automated formal verification of model transformations. In: *Proceeding of 2nd international workshop on critical systems development with UML (CSDUML 2003)*, pp 63–78
- [WE13] Wijs AJ, Engelen LJP (2013) Efficient property preservation checking of model refinements. In: *Proceeding of 19th international conference on tools and algorithms for the construction and analysis of systems (TACAS 2013)*, volume 7795 of LNCS. Springer, pp 565–579
- [WE14] Wijs AJ, Engelen LJP (2014) Refiner: towards formal verification of model transformations. In: *Proceeding of 6th NASA formal methods symposium (NFM 2014)*, volume 8430 of LNCS. Springer, pp 258–263
- [WF05] Wijs AJ, Fokkink WJ (2005) From χ_t to μ CRL: combining performance and functional analysis. In: *Proceeding of 10th conference on engineering of complex computer systems (ICECCS 2005)*. IEEE Computer Society Press, pp 184–193
- [Wij07] Wijs AJ (2007) Achieving discrete relative timing with untimed process algebra. In: *Proceeding of 12th conference on engineering of complex computer systems (ICECCS 2007)*. IEEE Computer Society Press, pp 35–44
- [Wij13] Wijs AJ (2013) Define, verify, refine: correct composition and transformation of concurrent system semantics. In: *Proceeding of 10th international symposium on formal aspects of component software (FACS 2013)*, volume 8348 of LNCS. Springer, pp 348–368
- [Wij15] Wijs AJ (2015) Confluence detection for transformations of labelled transition systems. In: *Proceeding of 2nd graphs as models workshop (GaM 2015)*, volume 181 of EPTCS. Open Publishing Association, pp 1–15
- [Win90] Winskel G (1990) A compositional proof system on a category of labelled transition systems. *Inf Comput* 87(1-2):2–57

Received 26 November 2016

Accepted in revised form 8 September 2017 by Perdita Stevens, Andrzej Wasowski, and Ewen Denney

Published online 9 October 2017