Define, Verify, Refine: Correct Composition and Transformation of Concurrent System Semantics

Anton Wijs[(✉)]

Department of Mathematics and Computer Science, Eindhoven University of Technology, 513, 5600 MB Eindhoven, The Netherlands
A.J.Wijs@tue.nl

Abstract. We present a technique to verify user-defined model transformations, in order to step-wise develop formal models of concurrent systems. The main benefit is that the changes applied to a model can be verified in isolation. In particular, the preservation of safety and liveness properties of such a modification can be determined independent of the input model. This is particularly useful for model-driven development approaches, where systems are designed and created by first developing an abstract model, and iteratively modifying this model until it is concrete enough to automatically generate source code from it. Properties that already hold on the initial model and should remain valid throughout the development in later models can be maintained with our tool Refiner, by which the effort of verifying those properties over and over again can be avoided. This paper generalises our earlier results in various ways, removing several restrictions, improving the focus of the verification method on transformations, and introducing the possibility to add completely new components at any time during the development.

1 Introduction

Concurrent systems tend to be very complex, and therefore very hard to develop correctly, i.e. bug-free. One approach to restrict the potential for introducing errors is by *step-wise* constructing the model of a concurrent system via model transformations. In that way, a model can be made more and more detailed, ultimately describing the system in full detail, which has the potential of allowing automatic source code generation. Such an approach can be made more robust by incorporating efficient verification techniques to determine that each intermediate model is correct, i.e. that desired functional properties are preserved. In [28,29], we presented a new technique to verify that formal definitions of transformations preserve desired functional properties, *independent* of the model they are applied on. Models, in this context, are action-based specifications of concurrent systems. Such specifications can be written in action-based modelling languages, such as process algebras. The definitions of transformations correspond with model transformations, as used in software engineering. The main benefit is that after application of a verified transformation, a model does not need to be rechecked, thereby avoiding state space explosion.

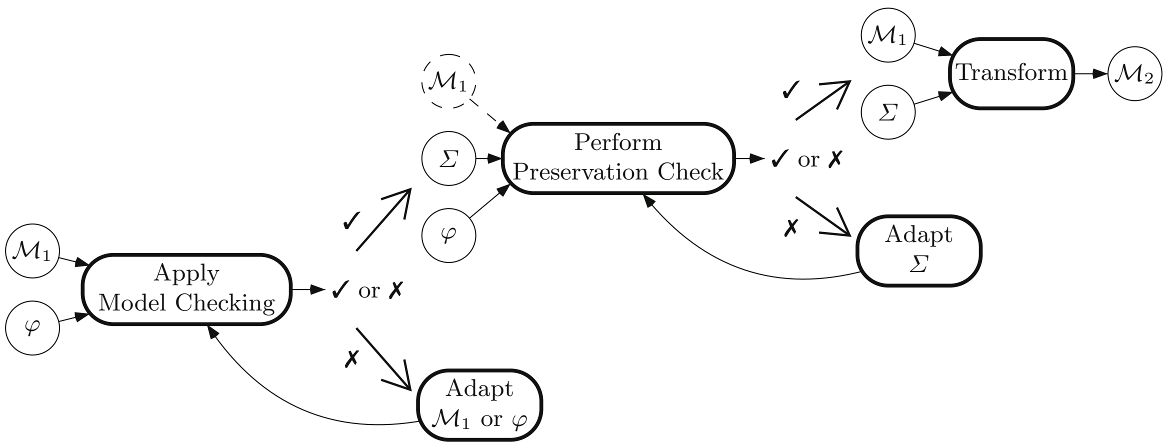


Fig. 1. Schematic depiction of the typical use of Refiner

The new verification technique has been implemented in a prototype tool called Refiner , by which a designer can, through a command-line interface, step-wise transform the semantics of processes in concurrent system designs. The designer does this by constructing transformation rule systems, which are formalisations of model transformations that can be analysed efficiently to determine whether they preserve safety or liveness properties *in general*, i.e. independent of the input model. Typically, Refiner is used as in Fig. 1. First, existing action-based model checking toolsets such as Cadp [8] and mCRL2 [5] can be applied to verify whether a given property φ, usually written in the modal μ-calculus [17], holds for model $\mathcal{M}$. If the property holds, Refiner is used to determine whether the property is preserved by rule system Σ. There are two supported techniques for this: one for determining *model-independent* property preservation, independent of input models, and one for determining property preservation for a particular input model. The latter involves $\mathcal{M}$ in the analysis, therefore it is not general, since it does not allow reusing the verification results for transformations of other models. But it may lead to a positive result in cases where the model-independent one does not. The model-independent property preservation check considers all the possibilities for Σ to match on input models. If φ is not preserved, Σ must be adapted and the last step repeated. If φ is preserved, Refiner can be used to transform model $\mathcal{M}$ into a model $\mathcal{M}$ satisfying φ.

Refiner is primarily a testbed to investigate the possibilities for verifying model transformations that exist, like models, as primary artifacts. This support should be non-intrusive, i.e. verification should be done in the background, hidden from the designer, in order to not burden him or her with the verification task.

Through experimentation, several limitations of the method from [29] have been identified. One is that the approach does not yet support *compositional* development of systems. Existing components can be transformed, but new

[1] Available for download at http://www.win.tue.nl/~awijs/refiner.

components cannot be introduced. Another is that a designer can sometimes be confronted by the limitations of the technique; in order for a rule system to be verifiable, it must be complete w.r.t. the behaviour it transforms. If two components can communicate, and one party is modified, then also the other party must be modified. This is not always desired, and requiring this makes the verification intrusive, and threatens scalability, because it may demand a large chain of additional modifications. Finally, we observe that in some cases, rule systems are constructed with particular models in mind. In such cases, checking property preserving for *all* imaginable models may be too restrictive.

Contribution. We build on the results of [29] to address the issues mentioned here. We enrich our transformation formalism with the ability to add new processes, and improve property preservation checking of rule systems. The latter is done by introducing a new construct for transformations, and by defining so-called *non-interface hiding*, which allows analysing the semantics of a subsystem w.r.t. the remainder of the system it is part of. Finally, Refiner exploits multi-core architectures through parallel property preservation checking, and we explain how this is achieved.

Roadmap. Section 2 discusses related work, and Sect. 3 introduces the notions used in this paper. In Sect. 4, property preservation checking from [29] is explained, and this is extended in various ways in Sect. 5. Our implementation and experimental results are shown in Sect. 6, and finally, Sect. 7 contains our conclusions and pointers to future work.

2 Related Work

Property preservation checking of changes applied on a model is most closely related to *incremental model checking* [23,24]. In that work, information about the verification computation is updated to reflect changes applied to the model. Most approaches are limited to checking safety properties, and all of them require at least as much computer memory as straight-forward model checking. Our technique, though, is also suitable for liveness properties and requires far less memory, since no information about the state space is maintained.

In *refinement checking* [1,18], supported by tools such as Rodin [2], FDR2 and Csp-Casl-Prover [16], 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 it not directly suitable to investigate the feasibility to verify definitions of model transformations, as opposed to the models they produce.

The Bart tool allows automatically refining B components to B0 implementations. Similar to our setting, it treats refinement rules as user-definable

[2] http://www.fsel.com/documentation/fdr2/html/index.html
[3] http://www.tools.clearsy.com/tools/bart

artifacts and performs pattern matching to do the refining. Constraints are checked to ensure that the resulting system will be correct. Approaches described in, e.g., [4,9,10,15] prove that a transformation preserves the semantics of any input model, by showing that the transformed model will be bisimilar to the original. Contrary to our work, in all these approaches, no form of automatic hiding of behaviour irrelevant for a desired system property is used, therefore they cannot handle cases where transformations alter the semantics in a way that does not invalidate that property. Others, such as [22,25], perform individual checks for each concrete model.

Finally, incremental system composition, as used by the tools Exp.Open [20] and Bip [3], focusses on incrementally combining processes into a full system, and the latter also provides a fixed number of correct-by-construction model transformations. With Refiner, one can define incremental process adding in terms of transformations, and it can verify transformations provided by the user. It will be interesting to see in how far results on compositional model checking can be reused, to further improve verification of such transformations.

3 Background

In this paper, the semantics of concurrent systems are defined in a compositional, action-based way. This means that the semantics of individual, finite-state processes are captured using *Labelled Transition Systems* (LTSs), and that these can be combined using synchronous composition, to obtain the semantics of a concurrent system as a whole. LTSs are action-based descriptions, indicating how a process can change state by performing particular actions.

An LTS $\mathcal{G}$ is a tuple $\langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G} \rangle$, where $\mathcal{S}_\mathcal{G}$ is a (finite) set of states, $\mathcal{A}_\mathcal{G}$ is a set of actions (including the invisible action τ), $\mathcal{T}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G} \times \mathcal{A}_\mathcal{G} \times \mathcal{S}_\mathcal{G}$ is a transition relation, and $\mathcal{I}_\mathcal{G} \subseteq \mathcal{S}_\mathcal{G}$ is a set of initial states. Actions in $\mathcal{A}_\mathcal{G}$ are denoted by a, b, c, etc. We use $s \xrightarrow{a}_\mathcal{G} s$ to denote $\langle s, a, s \rangle \in \mathcal{T}_\mathcal{G}$. If $s \xrightarrow{a}_\mathcal{G} s$, this means that in $\mathcal{G}$, an action a can be performed in state s, leading to state s.

Note that a state s can be interpreted as an LTS $\langle \{s\}, \emptyset, \emptyset, \{s\} \rangle$, and a transition $s \xrightarrow{a} s$ as an LTS $\langle \{s, s\}, \{a\}, s \xrightarrow{a} s, \{s\} \rangle$. We use underlining of states to indicate which states are initial, so, e.g., $\underline{s} \xrightarrow{a} \underline{s}$ represents $\langle \{s, s\}, \{a\}, s \xrightarrow{a} s, \{s, s\} \rangle$.

Network of LTSs. We represent models consisting of a finite number of finite-state concurrent processes by a number of LTSs and a set of *synchronisation laws*, or laws for short, defining how these LTSs interact. Together, these form a *network of LTSs* [20]. The process LTSs and laws imply a *system LTS*, representing the state space, which can be obtained by combining the LTSs using the laws. Given an integer $n > 0$, $1..n$ is the set of integers ranging from 1 to n. A vector $\overline{v}$ of size n contains n elements indexed by $1..n$. For $i \in 1..n$, $\overline{v}[i]$ denotes element i in $\overline{v}$.

[4] In [20], synchronisation laws are referred to as *rules*, but here, one may confuse these with transformation rules, that are introduced later in this section.

Definition 1 (Network of LTSs). *A network of LTSs $\mathcal{M}$ of size n is a pair $\langle \Pi, \mathcal{V} \rangle$, where*

- *Π is a vector of n (process) LTSs. For each $i \in 1..n$, we write $\Pi[i] = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \mathcal{I}_i \rangle$, and $s_1 \xrightarrow{b}_i s_2$ is shorthand for $s_1 \xrightarrow{b}_{\Pi[i]} s_2$;*
- *$\mathcal{V}$ is a finite set of synchronisation laws. A synchronisation law is a tuple $\langle \bar{t}, a \rangle$, where a is an action label, and $\bar{t}$ is a vector of size n called a* synchronisation vector*, in which for all $i \in 1..n$, $\bar{t}[i] \in \mathcal{A}_i \cup \{\bullet\}$, where $\bullet$ is a special symbol denoting that $\Pi[i]$ performs no action.*

At times, we use a set-notation for synchronisation vectors when the involved actions may appear in any order; e.g., for $n = 2$, $\{a\}$ denotes the set of vectors $\{\langle a, \bullet \rangle, \langle \bullet, a \rangle\}$. Furthermore, for $\langle \bar{t}, a \rangle$, $Ac(\bar{t}) = \{i \mid i \in 1..n \wedge \bar{t}[i] \neq \bullet\}$ refers to the set of processes active for $\langle \bar{t}, a \rangle$, and $A(\bar{t}) = \{\bar{t}[i] \mid i \in 1..n\} \setminus \{\bullet\}$ refers to the set of actions participating in $\langle \bar{t}, a \rangle$.

The synchronous composition of the LTSs in $\mathcal{M}$, i.e. the system LTS LTS($\mathcal{M}$), is the explicit description of the state space of the model. This LTS can be obtained by combining the behaviour of the $\Pi[i]$ according to the laws in $\mathcal{V}$:

- $\mathcal{I} = \{\langle s_1, \ldots, s_n \rangle \mid \forall i \in 1..n. s_i \in \mathcal{I}_i\}$, i.e. vectors of process initial states;
- $\mathcal{A} = \{a \mid \langle \bar{t}, a \rangle \in \mathcal{V}\}$, i.e. all actions that can result from synchronisation;
- $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, i.e. all possible combinations of process states;
- $\mathcal{T}$ is the smallest transition relation satisfying:

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

Example 1. Consider the two LTSs on the left in Fig. 2, in which the initial states are indicated by incoming arrowheads. We combine these in a network $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$, with Π containing those LTSs in order of appearance, and $\mathcal{V} = \{(\langle a, a \rangle, a'), (\langle b, b \rangle, b'), (\langle c, \bullet \rangle, c)\}$. The synchronous composition LTS($\mathcal{M}$) is displayed on the right in Fig. 2, where for each state, the ID pair in it indicates which combination of process LTS states it corresponds with. If both process LTS states have an outgoing a-transition, then so will the corresponding state in the synchronous composition. This also holds for b-transitions, but since b has data parameters, this only works if both occurrences have the same parameters d_1, d_2, which is the case here. This demonstrates how data can be used in transition labels, and how synchronisation works with it. Finally, the c-action can be fired independently, meaning that the first process LTS can move from state 2 to 3 without synchronisation.

Divergence-Sensitive Branching Bisimilarity. As equivalence relation between LTSs, we consider divergence-sensitive branching bisimilarity (DSBB) [11,12], which is sensitive to hidden behaviour and the branching structure of an LTS, including τ-cycles. Hence, it supports not only safety, but also liveness property preservation. For liveness properties, the notion of *diverging behaviour* is

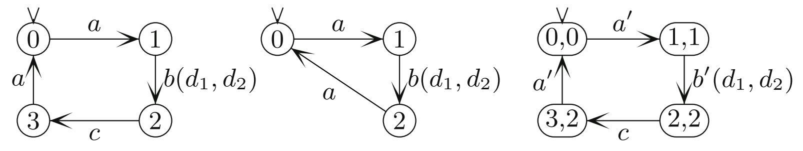


Fig. 2. Two LTSs and their synchronous composition (Example 1)

important. A state s is *diverging* iff an infinite sequence of internal actions can be performed, i.e. there exists an infinite τ-path from s, which for finite LTSs means that a τ-cycle is reachable via τ-transitions. We denote by $\rightarrow^+$ the transitive closure of $\xrightarrow{\tau}$.

Definition 2 (Divergence-Sensitive Branching Bisimulation). *A binary relation B between two sets of states $\mathcal{S}_{\mathcal{G}_1}$, $\mathcal{S}_{\mathcal{G}_2}$ of LTSs $\mathcal{G}_1$, $\mathcal{G}_2$ is a divergence-sensitive branching bisimulation if B is symmetric and $s\ B\ t$ with $s \in \mathcal{S}_{\mathcal{G}_1}$, $t \in \mathcal{S}_{\mathcal{G}_2}$ implies that*

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

– *if there is an infinite sequence of states $s_0, s_1, s_2, \ldots \in \mathcal{S}_{\mathcal{G}_1}$ such that $s_0 = s$, $s_0 \xrightarrow{\tau}_{\mathcal{G}_1} s_1 \xrightarrow{\tau}_{\mathcal{G}_1} s_2 \xrightarrow{\tau}_{\mathcal{G}_1} \ldots$ and $s_i\ B\ t$ for all $i \geq 0$, then there exists a $t' \in \mathcal{S}_{\mathcal{G}_2}$ such that $t \rightarrow^+ t'$ and $s_k\ B\ t'$ for some $k \geq 0$.*

Two states s and t are divergence-sensitive branching bisimilar, *noted $s \leftrightarrow_b^\Delta t$, if there is a divergence-sensitive branching bisimulation B with $s\ B\ t$.*

Two sets of states S, S' are DSBB, i.e. $S \leftrightarrow_b^\Delta S'$, iff $\forall s \in S. \exists s' \in S'. s \leftrightarrow_b^\Delta s'$ and vice versa. Two LTSs $\mathcal{G}_1, \mathcal{G}_2$ are DSBB, i.e. $\mathcal{G}_1 \leftrightarrow_b^\Delta \mathcal{G}_2$, iff $\mathcal{I}_{\mathcal{G}_1} \leftrightarrow_b^\Delta \mathcal{I}_{\mathcal{G}_2}$.

In [21], DSBB is related to a fragment of the modal μ-calculus, called L_μ^{dsbr}: if a model $\mathcal{M}_1$ satisfies an L_μ^{dsbr}-property φ, denoted by $\mathcal{M}_1 \models \varphi$, then a second model $\mathcal{M}_2$ satisfies φ iff $\mathcal{M}_1 \leftrightarrow_b^\Delta \mathcal{M}_2$. A similar result relates *branching bisimilarity* (BB) [12], i.e. DSBB without the divergence condition in Definition 2, and L_μ^{dsbr} safety properties, in which diverging behaviour is not relevant. In Sect. 4, we use this as follows: if we can determine that a transformation does not alter the system LTS structure, then we can conclude that φ will be preserved.

In [21,29], we actually also involve a hiding mechanism called *maximal hiding*, allowing to move LTSs to the highest possible level of abstraction w.r.t. a L_μ^{dsbr}-property φ. It involves rewriting transition labels not relevant for φ to τ, which roughly corresponds with hiding all labels not mentioned in φ. Incorporating this in property preservation checking makes the technique much more powerful, since it allows altering the semantics of a model through transformation, in ways not relevant for a given property. Given a network $\mathcal{M}_1$, let $H_\varphi(\text{LTS}(\mathcal{M}_1))$ be the maximally hidden synchronous composition of $\mathcal{M}_1$ w.r.t. property φ. Then, first of all, $\text{LTS}(\mathcal{M}_1) \models \varphi$ iff $H_\varphi(\text{LTS}(\mathcal{M}_1)) \models \varphi$, by maximal hiding [21]. Furthermore, by the relation between DSBB and L_μ^{dsbr}, if we can establish that

$H_\varphi(\mathrm{LTS}(\mathcal{M}_1)) \leftrightarrows_b^\Delta H_\varphi(\mathrm{LTS}(\mathcal{M}_2))$, then we can conclude that $H_\varphi(\mathrm{LTS}(\mathcal{M}_2)) \models \varphi$, and hence, that $\mathrm{LTS}(\mathcal{M}_2) \models \varphi$. In other words, it suffices to establish that the maximally hidden synchronous compositions are DSBB. For clarity, we only refer to hiding informally in some of the examples. It suffices to keep in mind that all labels not mentioned in the given property are hidden. For the specifics about L_μ^{dsbr}, the reader is referred to [21].

Transformation. In our setting, changes applied on a concurrent system model are represented by LTS *transformation rules* applied on the semantics of the processes of that model, i.e. on its network of LTSs. To reason about these changes, we define the notions of a rule, and matches of rules on process LTSs.

Definition 3 (Transformation Rule). *A* transformation rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *consists of a left pattern LTS* $\mathcal{L}^r = \langle \mathcal{S}_{\mathcal{L}^r}, \mathcal{A}_{\mathcal{L}^r}, \mathcal{T}_{\mathcal{L}^r}, \mathcal{I}_{\mathcal{L}^r} \rangle$ *and a right pattern LTS* $\mathcal{R}^r = \langle \mathcal{S}_{\mathcal{R}^r}, \mathcal{A}_{\mathcal{R}^r}, \mathcal{T}_{\mathcal{R}^r}, \mathcal{I}_{\mathcal{R}^r} \rangle$, *with* $\mathcal{I}_{\mathcal{L}^r} = \mathcal{I}_{\mathcal{R}^r} = (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r})$.

The states $\mathcal{I}_{\mathcal{L}^r}$ (and $\mathcal{I}_{\mathcal{R}^r}$) are called the *glue-states*, and they are all initial. They form the interface between behaviour subjected to transformation and the other behaviour. Process LTS states matched by glue-states will not be removed, but their incoming and outgoing transitions may be affected.

Definition 4 (Rule Match). *A transformation rule* $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ *has a* match $m_r : \mathcal{S}_{\mathcal{L}^r} \hookrightarrow \mathcal{S}_\mathcal{G}$ *on an LTS* $\mathcal{G} = \langle \mathcal{S}_\mathcal{G}, \mathcal{A}_\mathcal{G}, \mathcal{T}_\mathcal{G}, \mathcal{I}_\mathcal{G} \rangle$ *iff* m_r *is injective and*

1. $\forall s_1 \xrightarrow{a}_{\mathcal{L}^r} s_2 . m_r(s_1) \xrightarrow{a}_\mathcal{G} m_r(s_2)$;
2. $\forall s \in \mathcal{S}_{\mathcal{L}^r} \setminus \mathcal{I}_{\mathcal{L}^r}, p \in \mathcal{S}_\mathcal{G}$:
   - $m_r(s) \xrightarrow{a}_\mathcal{G} p \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s \xrightarrow{a} s' \wedge m_r(s') = p$;
   - $p \xrightarrow{a}_\mathcal{G} m_r(s) \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s' \xrightarrow{a} s \wedge m_r(s') = p$;
   - $m_r(s) = p \implies p \notin \mathcal{I}_\mathcal{G}$;

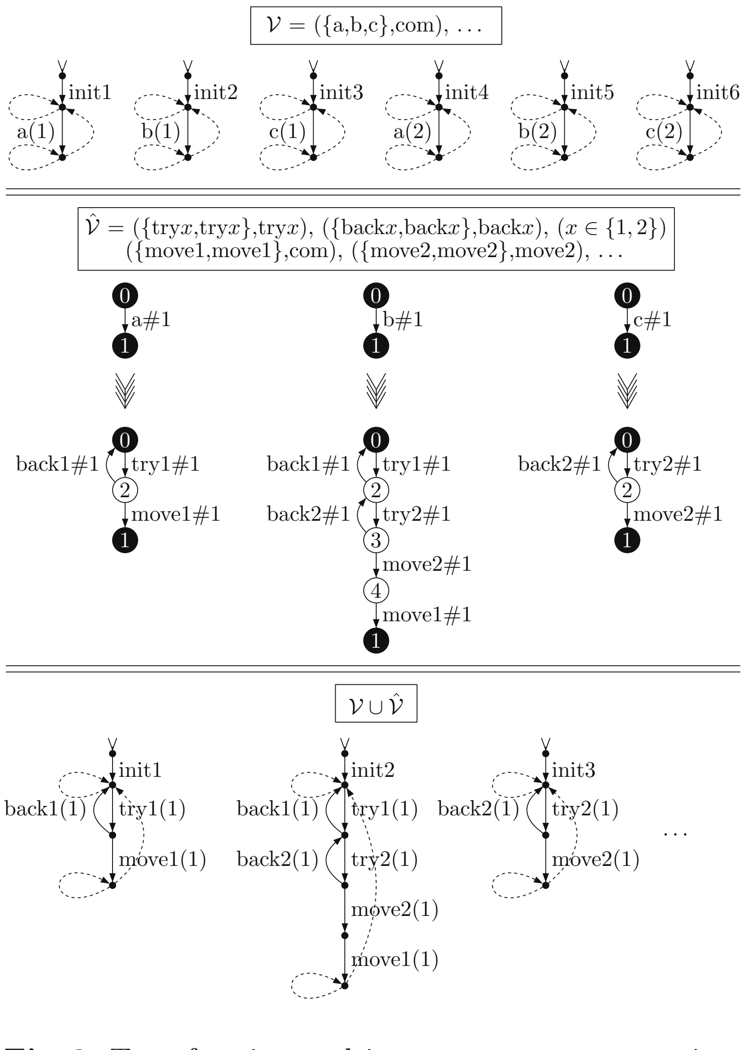
Note the conditions in the second clause of Definition 4. The first two are the *gluing conditions* of the *double-pushout* (DPO) method [14] for graph transformation, preventing conflicts when matching. They prevent so-called *dangling transitions*, which are transitions where only the source or target state will be removed, but not both. The final condition states that no initial state of $\mathcal{G}$ may be removed through transformation, ruling out the possibility of obtaining an LTS without an initial state.

When a left pattern is matched on part of a process LTS, transformation is performed by means of DPO. The result is that each state matched by a glue-state still exists after transformation, each state matched by a non-glue-state is removed, and each non-glue-state in a right pattern has resulted in appropriate representatives for each match of the left pattern.

In Sect. 5, we will introduce a form of *Negative Application Conditions* (NACs) [13]. The NACs of a rule express additional patterns that should *not* be matchable; a match can only be valid if the NAC patterns cannot be matched.

To facilitate explanation, we introduce a simplification without loss of generality. We assume that the $\mathcal{A}_i$ of the $\Pi[i]$ in $\mathcal{M}$ are disjoint. Any network for which this is not the case, e.g. the one given in Fig. 3, can be rewritten to one for which this holds. The simplification implies that for a rule system Σ, each rule $r \in R$ can only be applied on at most one process LTS. We use the convention that rule r_i can only be applied on process LTS $\Pi[i]$.

Sets of rules together make up a *rule system* $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$, with R a set of rules and $\hat{\mathcal{V}}$ a set of new synchronisation laws to be introduced when transforming. Transformation of a network of LTSs $\mathcal{M}$ according to a rule system Σ involves identifying all possible matches for each $r \in R$ on $\mathcal{M}$, and applying transformation on those matches. We say that $I_\Sigma = \{i \mid r_i \in R\}$. It represents the so-called *subsystem under transformation*; all $\Pi[i]$ with $i \in I_\Sigma$ are transformed by Σ.



$\mathcal{V}$ = ({a,b,c},com), ...
init1
a(1)
init2
b(1)
init3
c(1)
init4
a(2)
init5
b(2)
init6
c(2)
$\hat{\mathcal{V}}$ = ({try$x$,try$x$},try$x$), ({back$x$,back$x$},back$x$), ($x \in \{1,2\}$)
({move1,move1},com), ({move2,move2},move2), ...
a#1
b#1
c#1
back1#1
try1#1
move1#1
back2#1
try2#1
move2#1
$\mathcal{V} \cup \hat{\mathcal{V}}$
back1(1)
try1(1)
move1(1)
back2(1)
try2(1)
move2(1)
...

Fig. 3. Transforming multi- to two-party communication in a distributed system

Figure 3 shows an example of applying a rule system on a network of LTSs belonging to a distributed system design consisting of six processes. The behaviour of these processes relevant for our example is displayed at the top of the figure. After an initialisation step, each process can perform internal computations, represented by the unlabelled dashed transitions. At regular intervals, each process must synchronise with two others before commencing its computation. This is defined in the network by a law in $\mathcal{V}$: **({a,b,c},com)**. Actions a, b and c must have the same data parameter values for successful synchronisation, so only the first and the last three processes can potentially synchronise.

In the middle of Fig. 3, the definition of a rule system with three rules is displayed, and the glue-states are coloured black. Each rule is a pair of LTSs: the top one is the left pattern, and the bottom one is the right pattern.

The rule system of Fig. 3 defines how to break down the three-party synchronisation into a series of two-party synchronisations. To make rules more general, we use place-holders #1, #2, ... A place-holder in a left pattern represents that the parameters of a transition label can have any value, and the presence of the same place-holder in the corresponding right pattern indicates which transition labels should incorporate those values after transformation. In Fig. 3, the use of placeholders allows the rule system to be applicable on both the first three process LTSs and the last three. Additional laws in $\hat{\mathcal{V}}$ define the new synchronisation possibilities. This rule system is very practical if the system should eventually be able to run on hardware that does not support multi-party communication. Finally, part of the transformed network is displayed at the bottom of Fig. 3.

In the context of model transformations, it is crucial that a rule system is *terminating* and *confluent*, i.e. that the transformation is guaranteed to finish, and that it always leads to the same solution, independent of the order in which matches are processed. This is important, since a user defining how a particular model should be transformed typically has a specific resulting model in mind. Therefore, if a rule system is not confluent, it usually means that the user made some mistake. There are techniques to detect confluence, e.g. [19], which we have implemented. Here, we assume that a given rule system is confluent. Termination is achieved by the way in which we define transformation: first, all matches for all rules in the rule system are determined, and then, the rules are applied without looking for new matches. The process LTSs are finite, hence there will always be a finite number of matches.

4 Property Preservation Checking

The main contribution of our approach is the ability to efficiently check whether a rule system preserves desired functional properties, *without* analysing the potential behaviour of the input model. The verification techniques exploit the relations between DSBB and L_{μ}^{dsbr} properties, and BB and L_{μ}^{dsbr} safety properties on the one hand, and DSBB, BB, and maximal hiding on the other (see Sect. 3). Our techniques determine whether a rule system is guaranteed to preserve the structure of the synchronous composition of networks w.r.t. a property φ. This involves taking into account how the rule system can possibly be applied on networks, and checking for bisimilarities between combinations of dependent rule patterns, in which the possible synchronisation, and failure to synchronise, between rule patterns before and after transformation is analysed. The potential for synchronisation is derived from the laws and the $r_i \in R$, leading to sets of dependent rules, here referred to as *checks*. In general, a rule system can imply multiple checks. We say that Υ is the set containing all those checks. In order to compute Υ, we need a notion of *direct dependency* between rules. Behaviour in the rule patterns of r_i can directly depend on the behaviour of other rules. This is captured by the set $\delta(r_i)$. It is defined as:

$$\delta(r_i) = \bigcup_{\langle \bar{t}, a\rangle \in \mathcal{V} \cup \hat{\mathcal{V}}} \{r_j \in R \mid (\bar{t}[i] \in \mathcal{A}_{\mathcal{L}^{r_i}} \wedge \bar{t}[j] \in \mathcal{A}_{\mathcal{L}^{r_j}}) \vee (\bar{t}[i] \in \mathcal{A}_{\mathcal{R}^{r_i}} \wedge \bar{t}[j] \in \mathcal{A}_{\mathcal{R}^{r_j}})\}$$

Dependency is determined by the actions of the rule patterns, and the old and new laws. The *transitive closure* $\delta^{+}(r_i)$ contains all the rules on which r_i

depends, directly and indirectly. Essentially, a check consists of a set of dependent rules. Finally, we compute Υ as the set containing the $\delta^+(r_i)$ of all rules $r_i \in \Sigma$.

Example 2. In Fig. 3, let $\Pi[1], \ldots, \Pi[3]$ be the first three process LTSs at the top in order of appearance, and $r_1, \ldots, r_3$ be the rules in the middle in order of appearance. First of all, note that rule r_i is applicable on $\Pi[i]$, for $i \in \{1, 2, 3\}$. The relevant dependencies are $\delta^+(r_1) = \delta^+(r_2) = \delta^+(r_3) = \{r_1, r_2, r_3\}$. The same can be done concerning the other three process LTSs, by which we obtain the same set.

Before we formalise property preservation, we need to discuss one more issue. In order to correctly determine that a rule system preserves a given property, based on bisimilarities between vectors of left and right rule patterns, the rule patterns should be extended for analysis to make explicit which states are glue.

For example, consider a rule r that swaps two action labels a and b between two transitions, with $\mathcal{L}^r = \underline{s_0} \xrightarrow{a} \underline{s_1}, \underline{s_0} \xrightarrow{b} \underline{s_2}$, and $\mathcal{R}^r = \underline{s_0} \xrightarrow{b} \underline{s_1}, \underline{s_0} \xrightarrow{a} \underline{s_2}$. The two LTSs are DSBB, but only because s_1 of $\mathcal{L}^r$ (and of $\mathcal{R}^r$) can be related to s_2 of $\mathcal{R}^r$ (and of $\mathcal{L}^r$). However, both these states are glue, and hence can match on states of process LTSs that have in- and/or outgoing transitions that are not present in the patterns, and therefore may not be DSBB. This means that we are actually not interested in *any* DSBB, but a DSBB in which all glue-states in the left pattern are related to themselves in the right pattern. To express this, we add a self-loop with a unique action label to each glue-state in both patterns. Formally, for each glue-state s, we add a transition $s \xrightarrow{\kappa_s} s$, with κ_s the unique label. Since with this extension, each glue-state has at least one outgoing transition that no other state has, it has to be relatable to itself when trying to construct a DSBB. For the aforementioned example, the extended patterns, called $\mathcal{L}^r_\kappa$ and $\mathcal{R}^r_\kappa$, are not DSBB.

Adding *κ-loops* solves the problem of relating glue-states, but in practice, it turns out that it can be too restrictive. We return to this in Sect. 5, and introduce an improved way to extend patterns.

Each set $D \in \Upsilon$ defines two κ-extended vectors of LTSs $\overline{\mathcal{L}}^D_\kappa$, $\overline{\mathcal{R}}^D_\kappa$, where for $\mathcal{G} \in \{\mathcal{L}, \mathcal{R}\}$ and all $i \in 1..n$, we have $\overline{\mathcal{G}}^D_\kappa[i] = \mathcal{G}^{r_i}_\kappa$ if $r_i \in D$. In case $r_i \notin D$, we use a place-holder state in $\overline{\mathcal{G}}^D_\kappa$ at position i to indicate inactivity of r_i. The pairs $\langle \overline{\mathcal{L}}^D_\kappa, \overline{\mathcal{R}}^D_\kappa \rangle$ are used to check for property preservation. Together with the appropriate laws,[5] these vectors are interpreted as networks of LTSs, which therefore are implicit descriptions of system LTSs in which the synchronisation of process behaviour under transformation is described.

Finally, the property preservation check can be defined as follows.

Definition 5 (Property Preservation). *Given a network of LTSs $\mathcal{M}$, an L^{dsbr}_μ-property φ, and a rule system Σ, let Σ imply a set of rule sets Υ w.r.t. $\mathcal{V}$*

[5] Technically, the κ-actions require laws to produce κ-transitions in the synchronous composition of a network. For clarity, we do not include them in the formalisation.

and $\hat{\mathcal{V}}$. We say that Σ is φ-preserving *if for all $D \in \Upsilon$, $D' \subseteq D$, we have*

$$H_\varphi(LTS(\langle \overline{\mathcal{L}}_\kappa^{D'}, \mathcal{V}\rangle)) \leftrightarrow_b^\Delta H_\varphi(LTS(\langle \overline{\mathcal{R}}_\kappa^{D'}, \mathcal{V} \cup \hat{\mathcal{V}}\rangle))$$

In [6], a correctness proof is provided, i.e. that indeed, Σ is φ-preserving if the DSBB conditions hold.

Note that according to Definition 5, DSBB checks are required for *all subsets* D' of all $D \in \Upsilon$. Strict subsets of D represent situations where some processes are able to synchronise, but others are not. All these situations need to be checked, since they may occur in the system LTS of an input network.

Example 3. Say we want to check the preservation of deadlock freedom for the modification step defined in Fig. 3. In [21], it is explained that this allows to abstract from all transition labels, i.e. all can be rewritten to τ. Therefore, we can restrict checks to the (internal actions) branching structure of the rule patterns.

From Example 2, we know that the only relevant dependency is $\{r_1, r_2, r_3\}$. It implies two κ-extended vectors, containing the corresponding behaviour in the left and right rule patterns, respectively. Placing the vectors in two networks, combined with $\mathcal{V}$ and $\mathcal{V} \cup \hat{\mathcal{V}}$, we can compute two system LTSs. For the first (left patterns) network, we get the system LTS $L_1 = \underline{s_0} \xrightarrow{com\#1} s_1$ (ignoring the κ-loops), and for the second, we get LTS $L_2 = \underline{t_0} \xrightarrow{try1\#1} t_1 \xrightarrow{try2\#1} t_2 \xrightarrow{move2\#1} t_3 \xrightarrow{com\#1} t_4$, $t_1 \xrightarrow{back1\#1} \underline{t_0}$, and $t_2 \xrightarrow{back2\#1} t_1$ (again, ignoring the κ-loops). After hiding all transition labels, we find that $L_1 \not\leftrightarrow_b^\Delta L_2$, since L_2 contains τ-cycles and L_1 does not, but they are BB, hence deadlock freedom is preserved.

Definition 5 can be used to efficiently check for the property preservation by a rule system, thereby avoiding verification from scratch of the transformed model. However, a number of conditions regarding the applicability of a rule system were identified in [29].

1. *Universal applicability:* A rule system must be *universally applicable* w.r.t. actions subjected to synchronisation of at least two parties; if such an action a appears in the left pattern of some rule r_i, then *all* occurrences of a in $\Pi[i]$ must be matched on by that rule, i.e. all occurrences will be transformed. Without universal applicability, it is very hard to reason about the ability for the new network to synchronise, since the original and the transformed synchronising behaviour may coexist.
2. *Completeness:* A rule system must be *complete*, i.e. if one synchronising action is transformed, then all actions that it depends on must be transformed.
3. *Synchronisation:* Laws introduced through transformation can only involve new actions that were not present in the input model:

$$\forall \langle \bar{t}, a\rangle \in \hat{\mathcal{V}}, i \in 1..n. \bar{t}[i] \notin \bigcup_{i \in 1..n} \mathcal{A}_i$$

The reason for this is that otherwise, new laws can alter the semantics of a model in a way not expressed by the rules.

One contribution of this paper is a proposal how to remove the completeness condition entirely and relax the synchronisation condition. In addition, we introduce a mechanism to compositionally extend a network through transformation, a new hiding technique called *non-interface hiding*, allowing to focus the analysis on interfaces between subsystems, and the notion of an *exclusive* glue-state, which allows more expressiveness for defining rules.

5 Compositional Reasoning and Exclusivity NACs

Compositional Development. One major limitation of the setup in Sects. 3 and 4 is that it does not support adding new processes. This can be solved by interpreting network vectors as infinite vectors. Each vector can be considered to be infinite, with a finite number of process LTSs and an infinite number of 'placeholder' single states. For this, we define that for all $i > n$, $\Pi[i] = s_i$. Likewise, we interpret synchronisation vectors as being extended with an infinite number of $\bullet$-elements. Note that interpreting a network vector as being infinite does not affect its system LTS, as the additional processes never change state.

The extension allows the introduction of new process LTSs; a rule $r_i = \langle s, \mathcal{R}^{r_i} \rangle$, with $i > n$, effectively introduces a new process LTS isomorphic to $\mathcal{R}^{r_i}$ at position i in the network vector, since the single-state left pattern is applicable on the place-holder state at position i in the infinite vector. Note that a rule $\langle \mathcal{L}^{r_i}, s \rangle$ can be used to effectively remove a process LTS isomorphic to $\mathcal{L}^{r_i}$.

Removing the Completeness Condition. Another major limitation is the completeness condition of Sect. 4. Consider, for example, an input network with law $(\langle a, b \rangle, c)$, and we wish to transform transitions labelled b to transitions labelled b'. By the completeness condition, we would be forced to define a rule for a-transitions, even if we wish to keep these unchanged. This is not desired, since the verification technique is dictating how we should define a rule system.

Instead, we would like to be able to reason about behaviour subjected to transformation completely independent of behaviour that is not transformed. In the example, we would like to analyse a rule system applicable on b-transitions, without having to address the a-transitions. For this, we need to be able to focus our analysis entirely on the subsystem under transformation, and explicitly involve the potential for synchronisation with processes outside the subsystem, but not involve those processes themselves. Since synchronisation potential is represented by the synchronisation laws, this can be achieved by adding altered versions of laws that define synchronisation between the subsystem and the remainder of the system. The altered versions no longer require the remainder to be involved, thereby we detach the subsystem from the remainder of the system.

Definition 6 (Detaching Laws). *Given a network of LTSs $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ and a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$. We define the set of* detaching laws $\mathcal{V}_{det}$ *as follows:*

$$\mathcal{V}_{det} = \{\langle \bar{t}', \breve{a} \rangle \mid \langle \bar{t}, a \rangle \in \mathcal{V} \land (Ac(\bar{t}) \cap I_\Sigma) \neq \emptyset \land (Ac(\bar{t}) \setminus I_\Sigma) \neq \emptyset\},$$

with $\bar{t}'[i] = \bar{t}[i]$ for all $i \in I_\Sigma$, and $\bar{t}'[i] = \bullet$, otherwise, and $\breve{a}$ the action a annotated with the fact that it is the result of a detaching law.

For each law $\langle \bar{t}, a \rangle$ where some of the participating process LTSs $\Pi[i]$ will be transformed, i.e. $i \in Ac(\bar{t}) \cap I_\Sigma$, and some will not be, i.e. $i \in Ac(\bar{t}) \setminus I_\Sigma$, $\mathcal{V}_{det}$ contains a new law based on $\langle \bar{t}, a \rangle$, where behaviour of the latter process LTSs is ignored, and the behaviour of the former is kept. The set $\hat{\mathcal{V}}_{det}$ for the transformed network is defined in a similar way.

It is important to note that the actions resulting from laws in $\mathcal{V}_{det}$ (and $\hat{\mathcal{V}}_{det}$) should be excluded from maximal hiding. For this reason, those actions (the $\breve{a}$'s) have been annotated in Definition 6. When analysing the behaviour in the transformation rules, the potential for synchronisation between the subsystem under transformation, and thereby indirectly under analysis, and the remainder of the system, should be taken into account. We refer to this potential as the *interface* of the subsystem. The structure of the interface can be observed by hiding all actions except those that are the result of applying a detaching law. We refer to this as *non-interface hiding*, in which we move an LTS to a level of abstraction where we completely focus on the synchronisation with other LTSs.

Definition 7 (Non-Interface Hiding). *Given an LTS $\mathcal{G}$, the* non-interface hidden *LTS $H^{det}(\mathcal{G})$ is defined as follows:*

- $\mathcal{S}_{H^{det}(\mathcal{G})} = \mathcal{S}_\mathcal{G}$;
- $\mathcal{A}_{H^{det}(\mathcal{G})} = \{\breve{a} \mid \langle s, \breve{a}, s' \rangle \in \mathcal{T}_\mathcal{G}\} \cup \{\tau\}$;
- $\mathcal{T}_{H^{det}(\mathcal{G})} = \{\langle s, \breve{a}, s' \rangle \mid \langle s, \breve{a}, s' \rangle \in \mathcal{T}_\mathcal{G}\} \cup \{\langle s, \tau, s' \rangle \mid \langle s, a, s' \rangle \in \mathcal{T}_\mathcal{G}\}$;
- $\mathcal{I}_{H^{det}(\mathcal{G})} = \mathcal{I}_\mathcal{G}$.

Maximal hiding based on a property φ and non-interface hiding based on detaching laws can be combined into a general hiding technique that hides all actions except those that are relevant for the interface and/or the property. We denote this hiding by H^{det}_φ. Now, we redefine property preservation, taking the new notions into account.

Definition 8 (Improved Property Preservation). *Given a network of LTSs $\mathcal{M}$, an L^{dsbr}_μ-property φ, and a rule system Σ, let Σ imply a set of rule sets Υ w.r.t. $\mathcal{V}$ and $\hat{\mathcal{V}}$. We say that Σ is* φ-preserving *if for all $D \in \Upsilon$, $D' \subseteq D$, we have*

$$H^{det}_\varphi(LTS(\langle \overline{\mathcal{L}}^{D'}_\kappa, \mathcal{V} \cup \mathcal{V}_{det} \rangle)) \leftrightarrow^{\Delta}_{b} H^{det}_\varphi(LTS(\langle \overline{\mathcal{R}}^{D'}_\kappa, \mathcal{V} \cup \hat{\mathcal{V}} \cup \mathcal{V}_{det} \cup \hat{\mathcal{V}}_{det} \rangle))$$

Compared to Definition 5, the rule networks incorporate $\mathcal{V}_{det}$ and $\hat{\mathcal{V}}_{det}$, which allows for subsystems under transformation to be analysed in isolation.

For this new definition, the completeness condition can be dropped. However, for that to be useful, we need to relax the synchronisation condition. Otherwise, we would not be able to express new laws involving non-transformed behaviour.

The new condition is as follows: for all $\langle \bar{t}, a \rangle \in \hat{\mathcal{V}}$, there must exist a $\langle \bar{t}', a' \rangle \in \mathcal{V}$ such that for all $i \in 1..n \setminus I_\Sigma$, $\bar{t}[i] = \bar{t}'[i]$, and for all $i \in I_\Sigma$, both $\bar{t}'[i] \in \mathcal{A}_{\mathcal{L}^{r_i}} \cup \{\bullet\}$ and $\bar{t}[i] \in \mathcal{A}_{\mathcal{R}^{r_i}} \cup \{\bullet\}$. This expresses formally that the remainder of the system involved in the synchronisation was also allowed to synchronise in that setup in the original network, while the subsystem is allowed to be altered.

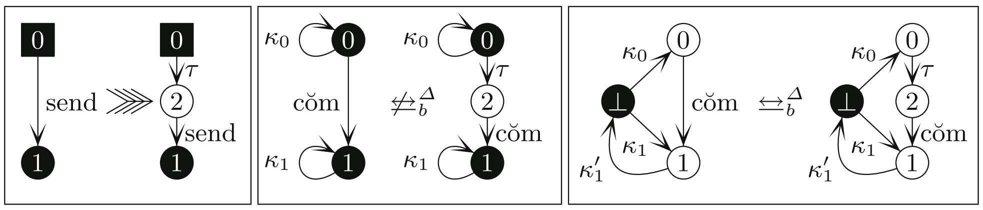


Fig. 4. κ-extension without and with exclusive glue-states

Example 4. Say we have a single rule r_2 with $\mathcal{L}^{r_2} = \underline{s_0} \xrightarrow{b} \underline{s_1}$, $\mathcal{R}^{r_2} = \underline{s_0} \xrightarrow{b'} \underline{s_1}$ in a rule system with $\hat{\mathcal{V}} = \{(\langle a, b' \rangle, c)\}$, that we wish to apply on a network $\mathcal{M}$ with $\mathcal{V} = \{(\langle a, b \rangle, c)\}$. By our convention, r_2 matches on $\Pi[2]$. Furthermore, by Definition 6, we have $\mathcal{V}_{det} = \{(\langle \bullet, b \rangle, \check{c})\}$ and $\hat{\mathcal{V}}_{det} = \{(\langle \bullet, b' \rangle, \check{c})\}$. Since there is only one rule, we can only construct check $\{r_2\}$. From this, by Definition 8, we obtain two networks $(\langle s, \mathcal{L}^{r_2}_{\kappa} \rangle, \{(\langle a, b \rangle, c), (\langle \bullet, b), \check{c} \rangle\})$ and $(\langle s, \mathcal{R}^{r_2}_{\kappa} \rangle,$ $\{(\langle a, b \rangle, c), (\langle \bullet, b \rangle, \check{c}), (\langle \bullet, b' \rangle, \check{c})\})$, with s a placeholder state. For both networks, the synchronous composition after H^{det}_{φ}-hiding is the LTS $\underline{s_0} \xrightarrow{\check{c}} \underline{s_1}$ with κ-loops for s_0 and s_1. The fact that the networks are DSBB indicates that both networks have the same potential for synchronisation with other system parts. This ensures that for a given L^{dsbr}_{μ}-property φ satisfied by $\mathcal{M}$, the transformed $\mathcal{M}$ satisfies φ as well.

Exclusivity NACs. Consider an LTS $L = \underline{s_0} \xrightarrow{compute} s_1 \xrightarrow{send} \underline{s_0}$, in which some computation is performed and the result is sent to another process using a law $(\{send, rec\}, com)$. Furthermore, consider that we wish to transform the *send*-transition through a rule system Σ containing a single rule with $\mathcal{L}^r = \underline{0} \xrightarrow{send} \underline{1}$ and $\mathcal{R}^r = \underline{0} \xrightarrow{\tau} 2 \xrightarrow{send} \underline{1}$. This rule is displayed on the left in Fig. 4. When analysing the rule patterns in isolation, which can be achieved by determining the sets of detaching laws, Σ turns out not to be property preserving for any φ, since added κ-loops prevent that, which is shown in the middle of Fig. 4. In particular, state 2 in the right LTS is not DSBB to state 0 on the left, due to the absence of an outgoing κ_0-transition. However, for our LTS L, the κ-loops do not truly represent the situation, since from state s_1, one can only perform a *send* action, but in the comparison, state 0, directly resulting from the glue-state matched on s_1, has other options, represented by the κ_0-transition.

To remove this limitation, we extend the notion of a rule further with *exclusive out* and *exclusive in/out* glue-states. Exclusive out glue-states are glue-states with the condition that they can only be matched on process LTS states for which all outgoing transitions are matched by the rule left pattern, i.e. they have no outgoing transition with a target state that is not matched. With exclusive out glue-states, a user can express that from particular states, one can only engage in matched behaviour, and not leave the pattern. In addition to this, exclusive in/out glue-states also have a similar condition for incoming transitions.

We extend the definition of a rule r with a set of exclusive out glue-states $\mathcal{E}^r_{out}$, and a set of exclusive in/out glue-states $\mathcal{E}^r_{in/out}$, with $\mathcal{E}^r_{in/out} \subseteq \mathcal{E}^r_{out} \subseteq \mathcal{I}_{\mathcal{L}^r}$ (and therefore, they are also subsets of $\mathcal{I}_{\mathcal{R}^r}$). We define $\mathcal{E}^r = \mathcal{E}^r_{out} \cup \mathcal{E}^r_{in/out}$.

These glue-states can be formalised using NACs. For a given rule $r = \langle \mathcal{L}^r, \mathcal{R}^r, \mathcal{E}^r_{out}, \mathcal{E}^r_{in/out} \rangle$, we add for each glue-state $s \in \mathcal{E}^r$ a NAC $s \overset{*}{\not\rightarrow} s'$ to a set of NACs $\mathcal{N}^r$, with s' a new state and $*$ a label place-holder indicating 'any label', and for each glue-state $s \in \mathcal{E}^r_{in/out}$, we add a NAC $s' \overset{*}{\not\rightarrow} s$, again with s' a new state and $*$ a label place-holder.

The patterns can be extended, taking exclusive glue-states into account.

Definition 9. *Given rule $r = \langle \mathcal{L}^r, \mathcal{R}^r, \mathcal{E}^r_{out}, \mathcal{E}^r_{in/out} \rangle$, the κ-*extended *$r_\kappa = \langle \mathcal{L}^r_\kappa, \mathcal{R}^r_\kappa, \mathcal{E}^r_{out,\kappa}, \mathcal{E}^r_{in/out,\kappa} \rangle$ is (re-)defined as follows:*

- *For $\mathcal{G} = \{\mathcal{L}, \mathcal{R}\}$, $\mathcal{G}^r_\kappa = \langle \mathcal{S}_{\mathcal{G}^r_\kappa}, \mathcal{A}_{\mathcal{G}^r_\kappa}, \mathcal{T}_{\mathcal{G}^r_\kappa}, \mathcal{I}_{\mathcal{G}^r_\kappa} \rangle$, where:*
  - $\mathcal{S}_{\mathcal{G}^r_\kappa} = \mathcal{S}_{\mathcal{G}^r} \cup \{\perp\}$
  - $\mathcal{A}_{\mathcal{G}^r_\kappa} = \mathcal{A}_{\mathcal{G}^r} \cup \{\kappa_s \mid s \in \mathcal{I}_{\mathcal{G}^r} \setminus \mathcal{E}^r_{in/out}\} \cup \{\kappa'_s \mid s \in \mathcal{I}_{\mathcal{G}^r} \setminus \mathcal{E}^r\}$
  - $\mathcal{T}_{\mathcal{G}^r_\kappa} = \mathcal{T}_{\mathcal{G}^r} \cup \{\langle \perp, \kappa_s, s \rangle \mid s \in \mathcal{I}_{\mathcal{G}^r} \setminus \mathcal{E}^r_{in/out}\} \cup \{\langle s, \kappa'_s, \perp \rangle \mid s \in \mathcal{I}_{\mathcal{G}^r} \setminus \mathcal{E}^r\}$
  - $\mathcal{I}_{\mathcal{G}^r_\kappa} = \mathcal{E}^r_{in/out} \cup \{\perp\}$
- $\mathcal{E}^r_{out,\kappa} = \mathcal{E}^r_{out}$
- $\mathcal{E}^r_{in/out,\kappa} = \mathcal{E}^r_{in/out}$

with $\perp$ a new initial state.

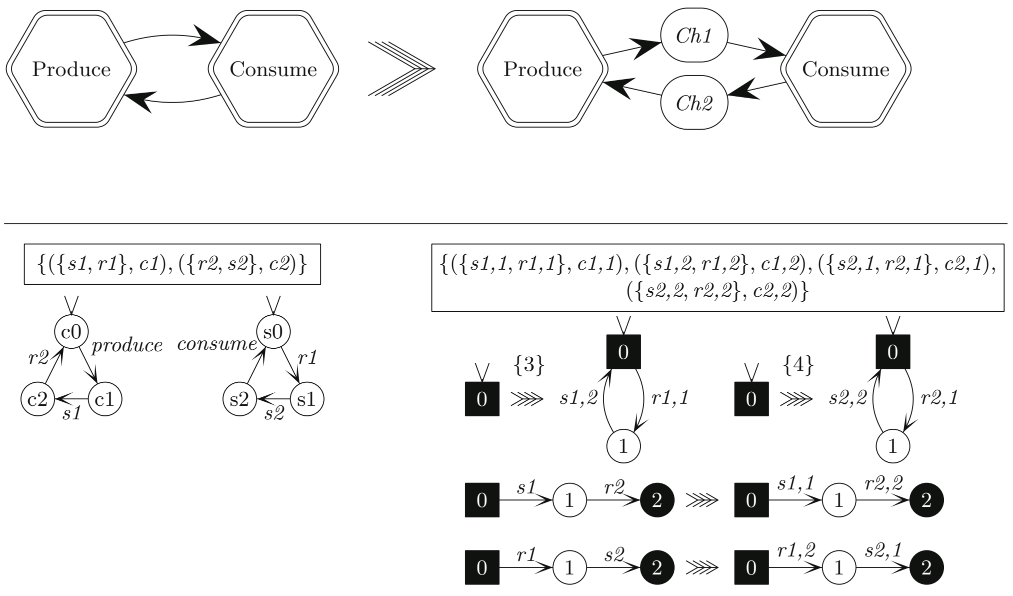
The new situation for our example is displayed on the right in Fig. 4, given that state 0 in the rule is an exclusive out glue-state (indicated by the fact that it is square in the rule on the left). The new state $\perp$ represents all the states in an LTS outside of a pattern match, and is used to formalise how a match of a pattern can relate to those states.

From $\perp$, the pattern can be entered via glue-states, and exited via non-exclusive glue-states. From state 0, one no longer has an alternative to performing the $c\breve{o}m$ (or τ)-transition, leading to the two LTSs being DSBB. In the next section, an example of using exclusive in/out glue-states is presented.

Correctness. The extensions presented in this section do not break the correctness of property preservation checking. First of all, exclusive glue-states can be handled in the proof by using the fact that such states do not have unmatched outgoing transitions (and incoming transitions, in the case of exclusive in/out glue-states). Second of all, the extensions concerning the detaching laws and the new synchronisation condition requires a more involved change to the proof.

Essentially, the extensions allow to determine that the LTS described by a subsystem under transformation in isolation is DSBB to the LTS described by the transformed subsystem. Since synchronisation with the remainder of the system can only be done via detaching laws, we know that both the original and the transformed subsystem will interact in the system in bisimilar ways, hence the overall system LTS maintains its structure.

An Example: Developing a Distributed System Finally, we demonstrate the use of the improvements presented in this section as part of our system development technique. We do this by means of an example of a producer-consumer system.



Produce
Consume
Ch1
Ch2
{({s1, r1}, c1), ({r2, s2}, c2)}
{({s1,1, r1,1}, c1,1), ({s1,2, r1,2}, c1,2), ({s2,1, r2,1}, c2,1), ({s2,2, r2,2}, c2,2)}
produce
consume
r2
r1
s1
s2
{3}
{4}
s1,2
r1,1
s2,2
r2,1
s1,1
r2,2
r1,2
s2,1

Fig. 5. Introducing channel components in a distributed system

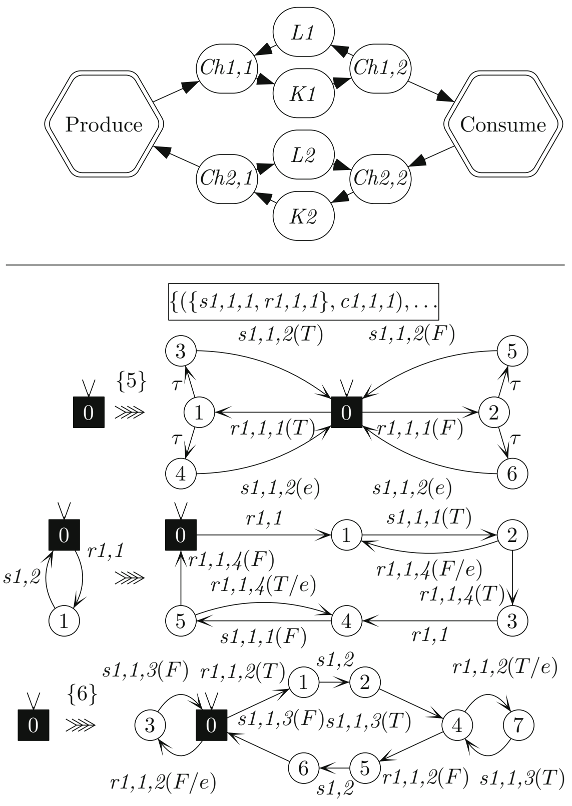


Fig. 6. Introducing ABP

In the left upper corner of Fig. 5, a schematic overview is given of a system in which the first component produces and sends a message, and the second one consumes that and sends a report back. We capture the semantics as displayed below the overview; two LTSs describe the components, in which the initial states are indicated by an incoming arrowhead, and two laws establish their synchronisation (the laws for *produce* and *consume* are not displayed).

In order to capture the use of channels more explicitly, we introduce two new components *Ch1*, *Ch2* through a rule system Σ_1. The new system is presented schematically to the right of the initial model, and rule system Σ_1 is displayed below that.

The extension to introduce new processes is crucial; the numbers above the transformation arrows indicate the IDs of the newly introduced process LTSs within the new network. Exclusive in/out glue-states are displayed as black, square states with an incoming arrowhead. They can accurately represent the initial states of newly introduced process LTSs, since those states neither have incoming, nor outgoing transitions that are not present in the pattern introducing the process LTS. Say we want to check the preservation of an L_{μ}^{dsbr}-property $\varphi = [\mathsf{true}^*]\ [produce]\ ([(\neg consume)^*]\ \neg\mathbf{deadlock} \wedge [\neg consume]\ \dashv)$, with **deadlock** $= [\mathsf{true}^*][\neg\tau]\mathsf{false} \wedge [\tau]\ \dashv$ expressing the presence of a deadlock. This expresses the inevitable reachability of a *consume* action after a *produce* action. After hiding all actions except for *produce* and *consume* [21], Σ_1 passes the check, i.e. relevant combinations of rule patterns lead to DSBB LTSs w.r.t. φ.

In the final step in Fig. 6, we introduce the Alternating Bit Protocol (ABP) for both channels, to reflect that in the final implementation, these channels will be lossy. Rule system Σ_2 consists of several rules, all of one of the three types that are displayed; the first one introduces a lossy channel, in this case *L1*, but *K1*, *L2*, and *K2* are introduced in a similar way. The second rule is used to transform *Ch1* into *Ch1,1*, which sends messages with an alternating bit to one lossy channel, and receives acknowledgements over the other channel.

Component *Ch2* is transformed similarly to *Ch2,2*. Finally, components *Ch1,2* and *Ch2,1* are introduced, which receive the messages, and, depending on the alternating bit, requests them to be resent, or forwards them to the other party. After hiding w.r.t. φ and detaching laws related to actions *s1,2*, *r1,1*, *s2,2*, and *r2,1*, i.e. the actions of the original channels that need to synchronise with the producer and consumer, Σ_2 does not preserve DSBB, since it introduces divergence, but it preserves BB, hence φ is preserved under the fairness condition that sending a message cannot fail infinitely often.

Note that in the final step, we do not have rules for the Consume and Produce components. Thanks to the detaching laws technique, we can analyse the introduction of ABP in isolation, without incorporating the remainder of the system. Without it, we would have to resort to analysing the whole system again, since all components are (indirectly) dependent on each other. Now, the largest LTS analysed contains 38 states, as opposed to an LTS of 1,720 states when performing the check without the detaching laws.

Finally, the use of exclusive glue-states is crucial. They accurately reflect the possible relation between matches of rule systems and the remaining states of process LTSs in general. This provides us with more potential to define transformations that can be verified in a model-independent way.

6 Implementation and Benchmark Results

Refiner is implemented in Python and can be run from the command-line. It is platform-independent, and allows performing behavioural transformations of networks of LTSs, and checking property preservation. It integrates with the action-based, explicit-state model checking toolsets Cadp [8] and mCRL2 [5]. These tools can be used to specify and verify concurrent systems. Refiner uses the mCRL2 tool LtsCompare to perform bisimilarity comparisons.

Definitions 5 and 8 indicate how property preservation checking could be performed in parallel; once the set of rule vectors Υ has been derived, system

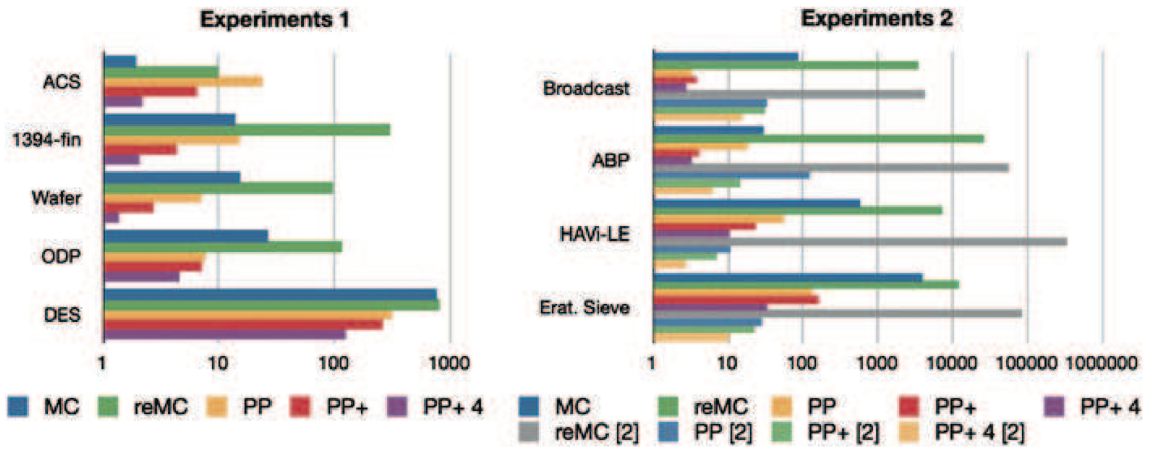


Fig. 7. Runtime comparisons (in seconds) of verification and property preservation checking. *(re)MC* = Model Checking (after transformation). *PP* = prop. pres. *PP+* = improved prop. pres. *PP+ 4* = 4-threaded PP+

LTSs must be constructed and compared for each (non-empty) subset of each $D \in \Upsilon$. The individual comparisons can be done independently of each other. Refiner can launch multiple comparison threads, thereby exploiting multi-core architectures.

We ran Refiner on a machine with a quad-core intel xeon E5520 2.27 GHz processor, 1 TB RAM, running Fedora 12. 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.[6] 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 synchronisations as in Fig. 3 (the broadcast and the HAVi leader election case). To give an indication of the state spaces sizes: the ACS case state space after transformation consists of about 22 thousand states, while after the second transformation, the HAVi-LE state space consists of 3 billion states. Relative to that, the verification runtimes are indicative of the sizes of the other state spaces.

Figure 7 compares runtimes for each model of verifying a property using the Cadp 2011-b tools Generator and Evaluator (this involves system LTS generation), and checking property preservation of the transformation using Refiner. Note the logarithmic scale. We performed one transformation per model for the experiments on the left, and two for those on the right ('[2]' indicates the runtimes after the second transformation).

The experiments demonstrate that preservation checking with Refiner is several 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 focusses on the applied change, not the resulting state space. Comparing the runtimes with those of other model checkers therefore leads to the same conclusion.

[6] The required files are available at http://www.win.tue.nl/~awijs/refiner.

Furthermore, the results demonstrate that the check with the improvements of Sect. 5 is often about 4 times faster than the original check, and linear speedups can be obtained on top of that with parallel checking. The parallel checks were performed using the four cores available on the test machine (*PP+ 4*), and further parallelisation is trivial. To give an indication of the number of bisimilarity checks performed, the largest number was 315 checks, for the first transformation in the ABP case.

7 Conclusions and Future Work

We presented a number of improvements of our property preservation checking technique for step-wise system development. Now, we are able to compositionally add new components and we have improved the ability to verify rule systems. With the new features, verification is made less intrusive to the designer, and she has more possibilities to step-wise construct her system through verified transformation steps.

As future work, we will continue to determine through experimentation whether there are more limitations in our technique that should be removed. Our final goal is to have a mature theory for verifying rule systems, and based on that, construct a model transformation language suitable for expressing verifiable transformation steps at the level of action-based modelling languages. This theory should also support timed behaviour, either using a timed version of bisimilarity, e.g. [7], or by modelling time in an untimed setting, e.g. [26]. Finally, possible applications of directed search techniques [27,30] will be investigated.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**, 253–284 (1991)
2. Abrial, J.-R., Butler, M., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
3. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the Bip framework. IEEE Softw. **28**(3), 41–48 (2011)
4. Blech, J.O., Glesner, S., Leitner, J.: Formal verification of Java code generation from UML models. In: Fujaba Days 2005, pp. 49–56 (2005)
5. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 199–213. Springer, Heidelberg (2013)
6. Engelen, L.J.P., Wijs, A.J.: Checking property preservation of refining transformations for model-driven development. CS-Report 12–08, TU Eindhoven (2012)
7. Fokkink, W.J., Pang, J., Wijs, A.J.: Is timed branching bisimilarity an equivalence indeed? In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 258–272. Springer, Heidelberg (2005)

8. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2010: a toolbox for the construction and analysis of distributed processes. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 372–387. Springer, Heidelberg (2011)
9. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards verified model transformations. In: 3rd International Workshop on Model Development, Validation and Verification (MoDeVVa 2006), pp. 78–93. IEEE Press, New York (2006)
10. Giese, H., Lambers, L.: Towards automatic verification of behavior preservation for model transformation via invariant checking. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2012. LNCS, vol. 7562, pp. 249–263. Springer, Heidelberg (2012)
11. van Glabbeek, R.J., Luttik, B., Trčka, N.: Branching bisimilarity with explicit divergence. Fundam. Inform. **93**(4), 371–392 (2009)
12. van Glabbeek, R.J., Weijland, W.P.: Branching time and abstraction in bisimulation semantics. J. ACM **43**(3), 555–600 (1996)
13. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundam. Inform. **26**(3–4), 287–313 (1996)
14. Heckel, R.: Graph transformation in a nutshell. Electron. Notes Theor. Comput. Sci. **148**, 187–198 (2006)
15. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, Ch., Wehrheim, H.: Showing full semantics preservation in model transformation - a comparison of techniques. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 183–198. Springer, Heidelberg (2010)
16. Kahsai, T., Roggenbach, M.: Property preserving refinement for Csp-Casl. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 206–220. Springer, Heidelberg (2009)
17. Kozen, D.: Results on the propositional μ-calculus. Theoret. Comput. Sci. **27**, 333–354 (1983)
18. Kundu, S., Lerner S., Gupta, R.: Automated refinement checking of concurrent systems. In: 26th International Conference on Computer-Aided Design (ICCAD 2007), pp. 318–325. IEEE Press, New York (2007)
19. Lambers, L., Ehrig, H.: Efficient conflict detection in graph transformation systems by essential critical pairs. Electron. Notes Theor. Comput. Sci. **211**, 17–26 (2008)
20. Lang, F.: Exp.Open 2.0: a flexible tool integrating partial order, compositional, and on-the-fly verification Methods. In: Romijn, J.M.T., Smith, G.P., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
21. Mateescu, R., Wijs, A.: Property-dependent reductions for the modal mu-calculus. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 2–19. Springer, Heidelberg (2011)
22. Narayanan, A., Karsai, G.: Towards verifying model transformations. Electron. Notes Theor. Comput. Sci. **211**, 191–200 (2008)
23. Sokolsky, O.V., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 351–363. Springer, Heidelberg (1994)
24. Swamy, G.M.: Incremental methods for formal verification and logic synthesis. Ph.D. thesis, University of California (1996)
25. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In: Critical Systems Development with UML (CSDUML 2003), pp. 63–78 (2003)
26. Wijs, A.J.: Achieving Discrete relative timing with untimed process algebra. In: 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), pp. 35–44. IEEE Press, New York (2007)

27. Wijs, A.J.: What to do next?: analysing and optimising system behaviour in time. Ph.D. thesis, VU University, Amsterdam (2007)
28. Wijs, A.J., Engelen, L.J.P.: Incremental formal verification for model refining. In: 9th International Workshop on Model Development, Validation and Verification (MoDeVVa 2012), pp. 29–34. ACM Press, New York (2012)
29. Wijs, A., Engelen, L.: Efficient property preservation checking of model refinements. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 565–579. Springer, Heidelberg (2013)
30. Wijs, A.J., Lisser, B.: Distributed extended beam search for quantitative model checking. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 166–184. Springer, Heidelberg (2007)