Efficient Property Preservation Checking of Model Refinements

Anton Wijs and Luc Engelen

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{A.J.Wijs,L.J.P.Engelen}@tue.nl

Abstract. In model-driven software development, models and model refinements are used to create software. To automatically generate correct software from abstract models by means of model refinement, desirable properties of the initial models must be preserved. We propose an explicit-state model checking technique to determine whether refinements are property preserving. We use networks of labelled transition systems (LTSs) to represent models with concurrent components, and formalise refinements as systems of LTS transformation rules. Property preservation checking involves determining how a rule system relates to an input network, and checking bisimilarity between behaviour subjected to transformation and the corresponding behaviour after transformation. In this way, one avoids generating the entire LTS of the new model. Experimental results demonstrate speedups of several orders of magnitude.

1 Introduction

Model-driven software development [2] entails creating implementations on a low level of abstraction from designs represented by models on a high level of abstraction. Implementation details, for example motivated by hardware restrictions, are added incrementally to these abstract models by means of refining model transformations. Usually, an implementation must satisfy a number of requirements that can be expressed as properties of the model that forms its design. Then, the transformations should preserve these properties. Model checking [4] can help to determine whether this is the case, but verifying the properties from scratch for each new model along the development chain not only requires much time, but it is also likely to become unfeasible very quickly, as the related state space of a model tends to grow exponentially when applying a refinement.

In this paper, we present an explicit-state model checking technique tailored for incremental refinement of models of concurrent systems. If the model that forms the initial design of such a system is relatively small, then at this stage, properties can still be verified using traditional techniques based on explicit state space exploration. When a refinement needs to be applied, then instead of the refined model, the technique analyses the formal semantics of the *refinement*, and determines whether application of the refinement is guaranteed to preserve

N. Piterman and S. Smolka (Eds.): TACAS 2013, LNCS 7795, pp. 565–579, 2013.
© Springer-Verlag Berlin Heidelberg 2013

a particular property. This can be either a safety, liveness, or fairness property. This *property-preservation checking* is purely done by reasoning about the structures of Labelled Transition Systems (LTSs), which we use to express the semantics of both the models and the refinements. For a model, the semantics of each process in the model is expressed as an LTS, and the semantics of the complete concurrent system is expressed implicitly by a *network* of LTSs [19], describing how these process LTSs interact. A refinement is formalised as a system of *LTS transformation rules*, where each rule has a left LTS pattern, describing what should be changed, and a right LTS pattern, describing what the result of the change should be. Furthermore, such a system can add to the interaction mechanism of the processes. By focussing on LTSs, our technique is applicable to any modelling language, either to describe models or refinements, whose semantics can be expressed by LTSs.

Mateescu and Wijs [21] developed an automatic technique called *maximal hiding*, which works for a particular, but still very expressive, fragment of the modal μ-calculus [17]. It identifies all system behaviour not relevant for a given property, and hides all this behaviour, i.e. renames the transition labels to τ. Furthermore, it is compatible with *divergence-sensitive branching bisimilarity* (DSBB) [11]. DSBB is a useful equivalence that respects branching-time and cycles of internal behaviour, and is therefore not only suitable for safety properties, but also liveness and fairness properties. The compatibility lies in the fact that if two LTSs are maximally hidden w.r.t. the same property, and they are DSBB, then they both do or do not satisfy the property. By identifying *all* irrelevant behaviour, maximal hiding maximises the potential for a positive DSBB comparison result.

We use these results to focus on the following question: given a model $\mathcal{M}$ satisfying a property φ written in the μ-calculus fragment, and given a system of transformation rules Σ, when and how can we determine whether it is guaranteed that Σ will not structurally alter the maximally hidden LTS of $\mathcal{M}$ when it is applied to it, i.e. will the resulting LTS of the new model $\mathcal{M}'$ be DSBB to the one of $\mathcal{M}$, if both are maximally hidden? As it turns out, this can be done without investigating the LTS of $\mathcal{M}'$ if some reasonable conditions regarding the applicability of Σ on $\mathcal{M}$ are met.

Shifting the focus from models to model refinements implicitly assumes that a modeller likewise focusses on defining refinements to move her initial model to increasingly lower levels of abstraction, and it is in these refinements where she can influence the development. One could also imagine building a dictionary of reusable refinement patterns, including a pattern to, for example, add functionality to cope with lossy communication channels. In our experimental section, we demonstrate that our technique is applicable for such refinements, in fact it runs several orders of magnitude faster than verifying the refined model.

This paper is structured as follows. Section 2 introduces the preliminaries. In Section 3, we formalise LTS transformation. Next, in Section 4, we discuss our technique for determining whether transformations preserve properties. Experimental results are given in Section 5. Section 6 discusses related work, and Section 7 contains conclusions and pointers to future work.

2 Preliminaries

Labelled transition system. 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 $s_1 \xrightarrow{a}_\mathcal{G} s_2$ as a shorthand for $\langle s_1, a, s_2 \rangle \in \mathcal{T}_\mathcal{G}$. If $s_1 \xrightarrow{a}_\mathcal{G} s_2$, this means that in $\mathcal{G}$, an action a can be performed in state s_1, leading to state s_2.

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 rules* defining how these LTSs interact. For this, we use the concept of *networks of LTSs* [19]. 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 $\overline{v}$.

Definition 1. *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 rules. A synchronisation rule is a tuple $\langle \overline{t}, a \rangle$, where a is an action label, and $\overline{t}$ is a vector of size n called a* synchronisation vector*, in which for all $i \in 1..n$, $\overline{t}[i] \in \mathcal{A}_i \cup \{\bullet\}$, where $\bullet$ is a special symbol denoting that $\Pi[i]$ performs no action.*

With $\mathcal{A}_{1..n}$, we refer to the union of the $\mathcal{A}_i$. Furthermore, for $\langle \overline{t}, a \rangle$, $Ac(\overline{t}) = \{i \mid i \in 1..n \wedge \overline{t}[i] \neq \bullet\}$ refers to the set of processes active for $\langle \overline{t}, a \rangle$, and $A(\overline{t}) = \{\overline{t}[i] \mid i \in 1..n\} \setminus \{\bullet\}$ refers to the set of actions participating in $\langle \overline{t}, a \rangle$.

A network of LTSs $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$ is an implicit description of all possible system behaviour of the model. We call the explicit description the *system LTS* $\langle \mathcal{S}_\mathcal{M}, \mathcal{A}_\mathcal{M}, \mathcal{T}_\mathcal{M}, \mathcal{I}_\mathcal{M} \rangle$. It can be obtained by combining the $\Pi[i]$ according to the rules in $\mathcal{V}$:

- $\mathcal{I}_\mathcal{M} = \{\langle s_1, \ldots, s_n \rangle \mid \forall i \in 1..n . s_i \in \mathcal{I}_i\}$;
- $\mathcal{A}_\mathcal{M} = \{a \mid \langle \overline{t}, a \rangle \in \mathcal{V}\}$;
- $\mathcal{S}_\mathcal{M} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$;
- $\mathcal{T}_\mathcal{M}$ is the smallest transition relation satisfying: $\langle \overline{t}, a \rangle \in \mathcal{V} \wedge \forall i \in 1..n . (\overline{t}[i] = \bullet \wedge s'[i] = s[i]) \vee (\overline{t}[i] \neq \bullet \wedge s[i] \xrightarrow{\overline{t}[i]}_i s'[i]) \implies s \xrightarrow{a}_\mathcal{M} s'$.

On the left of Figure 1, a network consisting of three process LTSs and four synchronisation rules is shown. The black states are initial. The network LTS representing the behaviour of this network is shown on the right. The figure demonstrates the expressiveness of networks of LTSs. It shows, for example, that multi-party synchronisation is offered, as illustrated with the synchronisation rule $\langle \langle f, f, f \rangle, f \rangle$. This rule specifies that action f in the system LTS is the result of the synchronisation of the actions f of the three processes. Rule $\langle \langle b, d, \bullet \rangle, e \rangle$ specifies a synchronisation between processes $\Pi[1]$ and $\Pi[2]$, rule $\langle \langle a, \bullet, \bullet \rangle, a \rangle$ specifies that action a of process $\Pi[1]$ can be executed independently, and rule $\langle \langle \bullet, c, \bullet \rangle, c \rangle$ specifies the same for action c of process $\Pi[2]$.

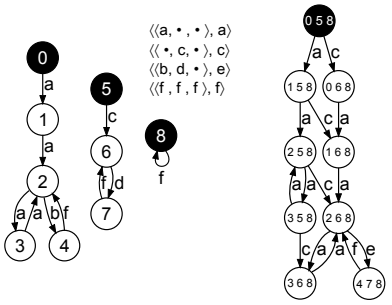

Fig. 1. A network of LTSs and its system LTS

Synchronisation rules can also be used to introduce non-deterministic behaviour, by specifying multiple rules for the same actions. By adding the rule $\langle\langle a, c, \bullet\rangle, g\rangle$, $\Pi[1]$ and $\Pi[2]$ can either synchronise on a and c, or perform them independently.

Hiding. To abstract from certain actions in networks, we define the hiding operator τ_H, which renames all actions in action set H, i.e. the *hiding set*, to τ: $\tau_H(\mathcal{M}) = \langle \Pi, \{(\langle \overline{t}, a\rangle \in \mathcal{V} \mid a \notin H\} \cup \{\langle \overline{t}, \tau\rangle \mid \langle \overline{t}, a\rangle \in \mathcal{V} \wedge a \in H\}\rangle$. Intuitively, hidden behaviour should neither be subjected to synchronisation, nor renamed: $\forall \langle \overline{t}, a\rangle \in \mathcal{V}.\tau \in A(\overline{t}) \implies |Ac(\overline{t})| = 1 \wedge a = \tau$. Hidden behaviour should also always be enabled: $\forall i \in 1..n.\exists \langle \overline{t}, \tau\rangle \in \mathcal{V}.A(\overline{t}) = \{\tau\} \wedge Ac(\overline{t}) = \{i\}$. We only consider networks for which these conditions hold.

Maximal Hiding. Mateescu and Wijs [21] explained how to derive for LTS $\mathcal{G}$ and temporal logic formula φ the *largest possible* hiding set $h_{\mathcal{A}_\mathcal{G}}(\varphi)$, if φ is written in a fragment of the modal μ-calculus [17] called L_μ^{dsbr}. Hiding this set, i.e. applying maximal hiding, allows moving to the highest possible level of abstraction without disturbing the truth-value of φ. L_μ^{dsbr} can express safety, liveness, and fairness properties. We denote the maximally hidden LTS $\mathcal{G}$ w.r.t. φ by $\tilde{\tau}_\varphi(\mathcal{G})$.

Divergence-Sensitive Branching Bisimulation. To compare LTSs, we use the equivalence relation *divergence-sensitive branching bisimulation* (DSBB) [11]. It supports hidden behaviour, and is sensitive to the branching structure of an LTS, including τ-cycles. Hence besides safety properties, it also supports fairness (e.g. livelock), and liveness properties. We call a state s *diverging*, iff an infinite τ-path is reachable from s. For finite LTSs, this means that a τ-cycle is reachable via τ-transitions. A formal definition of DSBB is not required for the understanding of this paper; the interested reader is referred to [11].

DSBB is compatible with maximal hiding. This allows reasoning about LTSs w.r.t. properties. Given a L_μ^{dsbr} formula φ and LTSs $\mathcal{G}_1$ and $\mathcal{G}_2$, if we know that $\mathcal{G}_1$ satisfies φ, we can conclude whether or not $\mathcal{G}_2$ satisfies φ by applying maximal hiding on both LTSs, and determining whether $\tilde{\tau}_\varphi(\mathcal{G}_1)$ is DSBB to $\tilde{\tau}_\varphi(\mathcal{G}_2)$. Based on this, our proposed preservation check, formulated in Section 4, determines whether an LTS *transformation* application possibly alters the branching structure of an LTS. If not, we can conclude that φ is preserved after transformation.

3 LTS Transformations

In this section, we formalise refinement steps as transformations of networks of LTSs. A network is transformed by transforming the individual process LTSs that constitute it and adding additional synchronisation rules.

3.1 Transformation Rules

LTSs are transformed by applying transformation rules. These rules are defined as follows.

Definition 2. *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* $\mathcal{I}_{\mathcal{L}^r} = \mathcal{I}_{\mathcal{R}^r} = (\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r})$.

States $\mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r}$, also referred to as the glue-states, are all initial and define how $\mathcal{R}^r$ should replace $\mathcal{L}^r$. All changes to an LTS are applied relative to these glue-states. We call a rule $r = \langle \mathcal{L}^r, \mathcal{R}^r \rangle$ applicable on an LTS $\mathcal{G}$ iff there exists a match $m_r : \mathcal{S}_{\mathcal{L}^r} \hookrightarrow \mathcal{S}_{\mathcal{G}}$ (an embedding) for which the following holds:

Definition 3. *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. $\forall s \in \mathcal{S}_{\mathcal{L}^r} \setminus \mathcal{S}_{\mathcal{R}^r}, p \in \mathcal{S}_{\mathcal{G}}$:
   - $m_r(s) = p \implies \neg \exists s' \in \mathcal{S}_{\mathcal{L}^r} \cap \mathcal{S}_{\mathcal{R}^r} . m_r(s') = p$;
   - $m_r(s) \xrightarrow{a}_{\mathcal{G}} p \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s \xrightarrow{a} s' \wedge m_r(s') = p$;
   - $p \xrightarrow{a}_{\mathcal{G}} m_r(s) \implies \exists s' \in \mathcal{S}_{\mathcal{L}^r} . s' \xrightarrow{a} s \wedge m_r(s') = p$.

The second point of Definition 3 expresses the *gluing conditions* [14]. The first condition, the *identification condition*, says that in a single match, there may not be a contradiction concerning the removal of states, which could happen if both a glue-state and a non-glue-state are matched on the same state. The remaining two points express the *dangling condition*, which rules out the removal of transitions in $\mathcal{G}$ that are not explicitly represented in $\mathcal{L}^r$, i.e. it is not allowed that only one state of a transition is matched on and scheduled to be removed. We follow the double-pushout approach (DPO) [7], i.e. if the gluing condition is violated, the match is not valid. If we would apply transformation on a match even though the condition is violated, then the effect would be unpredictable, which would also limit our ability to reason about the structure of the result.

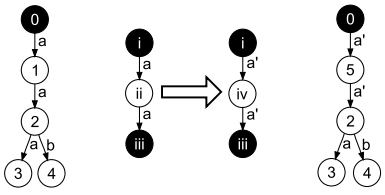

Fig. 2. Rule matching

In the middle of Figure 2, a transformation rule is shown. All initial and glue-states are coloured black in this figure. The rule defines that any state matched on state *ii* of the left pattern of the rule should be removed and replaced by a new state, which is labeled *iv* in the rule. Therefore, the left pattern can be matched on states $\{0, 1, 2\}$ of the LTS on the left of the figure, but not on states $\{1, 2, 3\}$. The latter match would result in the removal of state 2 and lead to a dangling transition.

Transformation of a network of LTSs proceeds as follows: First, the largest set of matches for a rule on each process LTS is determined. Then, for each match,

DPO is applied to replace left pattern matches by copies of the right pattern, i.e. first remove all states and transitions matched by $\mathcal{L}^r \setminus \mathcal{R}^r$, and then place a copy of $\mathcal{R}^r \setminus \mathcal{L}^r$ in the result. Using this approach, termination of transformation is no issue; we do not recompute the set of matches for intermediate transformation results, and since the $\Pi[i]$ are finite, there is a finite number of matches initially.

Figure 2 illustrates the application of a transformation rule. The LTS on the right is the result of applying the rule in the middle to the LTS on the left.

3.2 Rule Systems

With transformation rules, a *rule system* $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ can be built, with R a set of transformation rules and $\hat{\mathcal{V}}$ a set of synchronisation rules to be introduced in the result of a transformation. Transformation of a network of LTSs via a rule system is done by determining for every $\Pi[i]$ and every rule the set of all matches, and applying transformation on these matches.

Here, a rule system defines how a network of LTSs should be transformed into a more refined network. In that context, it is important that a rule system is confluent, i.e. application always produces the same result. When a user defines a transformation, she desires to obtain a single, refined model. From graph theory, it is known that confluence is undecidable for general rule systems, but it is decidable under certain conditions [18]. Here, we ensure that a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$ is confluent for an LTS $\mathcal{G}$ by 1) requiring that the action sets of left patterns of rules are disjoint, i.e. $\forall r_1, r_2 \in R. \mathcal{A}_{\mathcal{L}^{r_1}} \cap \mathcal{A}_{\mathcal{L}^{r_2}} = \emptyset$, and 2) checking for each $\Pi[i]$ that no two matches of a single rule intersect, i.e. $\forall r \in R. \neg\exists m_r^1, m_r^2, s_1, s_2 \in \mathcal{S}_{\mathcal{L}^r}. m_r^1 \neq m_r^2 \wedge m_r^1(s_1) = m_r^2(s_2)$. By 1) and the dangling condition, transformation of a match of one rule cannot influence a match of another rule, and by 2), neither can it influence the matches of the same rule. The first condition can efficiently be checked before matches are determined, and the second can be done while matches are determined. Note that these restrictions are more strict than technically required, but they still allow efficient confluence checking. If a rule system is not confluent, it often indicates that the user overlooked something; if not, then usually a confluent rule system can be obtained by e.g. rewriting actions, merging rule patterns, and / or splitting rule systems in multiple ones.

4 Checking Property Preservation

Our property preservation check for rule systems actually entails a number of computations and checks, which have been implemented in a new tool called Refiner. The only required input is a network of LTSs and a rule system, specified by the user. Figure 3 gives an overview of the approach.

Given $\mathcal{M}$ and Σ, the tool takes the following steps, which will be explained in more detail in this section:

1. Check that the new synchronisation rules $\hat{\mathcal{V}}$ are *well-formed* w.r.t. $\mathcal{M}$;
2. Generate a set Υ of *rule sets*;

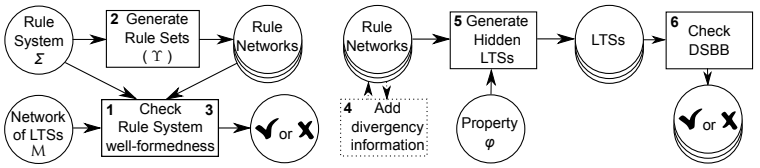

Fig. 3. Checking well-formedness and property preservation of a rule system

3. Check that Υ only contains *well-formed* sets w.r.t. $\mathcal{M}$;
4. Optionally, add divergency information to the rule sets;
5. For each rule set $\rho \in \Upsilon$, generate pairs of corresponding system LTSs and apply maximal hiding;
6. For each pair of LTSs, perform a DSBB comparison. If all DSBB comparisons in the previous step were positive, then Σ preserves φ.

We introduce two simplifications to facilitate explanation. First of all, we assume that in $\mathcal{M} = \langle \Pi, \mathcal{V} \rangle$, all the $\mathcal{A}_i$ are disjoint. This is not a fundamental limitation, since renaming of actions and modifying the synchronisation rules can enforce this. For a rule system $\Sigma = \langle R, \hat{\mathcal{V}} \rangle$, this implies that each $r \in R$ can be applicable on at most one process LTS. If a similar transformation must be applied to multiple process LTSs, including in Σ multiple copies of a rule with appropriately renamed actions suffices. Based on this, we define a function $I : 2^{\mathcal{A}_{1..n}} \times \mathbb{N} \to 2^{\mathcal{A}_i}$, with, given an action set A, $I(A, i) = A \cap \mathcal{A}_i$.

Second of all, for $\mathcal{M}$ and Σ, we assume that each $\Pi[i]$ is matched on by exactly one r. This is expressed by indexing the $r \in R$ such that rule r_i is matched on $\Pi[i]$. This is also not a real limitation; if multiple rules are applicable on a $\Pi[i]$, Σ can be rewritten since it is confluent. This is done by splitting the rule system into multiple ones, and applying these one after the other.

1. Well-formedness of $\hat{\mathcal{V}}$. We restrict the ability to introduce new synchronisation rules in order to determine property preservation. Otherwise, by defining new synchronisation rules over already existing actions, a model could be altered without actually defining any transformation rules. We check that each rule in $\hat{\mathcal{V}}$ only contains actions in its vector that are introduced by Σ:

$$\forall \langle \overline{t}, a \rangle \in \hat{\mathcal{V}}, i \in 1..n. \overline{t}[i] \in (\mathcal{A}_{\mathcal{R}^{r_i}} \setminus \mathcal{A}_i) \cup \{\bullet\}$$

This does not limit the ability to express transformations, however; e.g. if two existing actions a and b should synchronise after transformation, one can define two transformation rules renaming these to a' and b', respectively, and define a new synchronisation rule for these new actions.

2. Generate Rule Sets. The synchronisation rules of $\mathcal{M}$ directly give rise to a dependency function δ for actions in the $\mathcal{A}_i$, where for each b, we have $\delta(b) = \bigcup_{\langle \overline{t}, a \rangle \in \mathcal{V}} \{\overline{t}[j] \mid j \in 1..n \wedge b \in \mathcal{A}_i \wedge \overline{t}[i] = b\} \setminus \{\bullet\}$. It defines the set of actions on

which b depends to be able to synchronise in $\mathcal{M}$. This function can be used to identify the set of dependent actions containing the action set of the left pattern of an $r_i \in R$; it is the smallest closed set $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ of $\mathcal{A}_{\mathcal{L}^{r_i}}$ w.r.t. δ and the subset relation, i.e. $\mathcal{A}_{\mathcal{L}^{r_i}} \subseteq C(\mathcal{A}_{\mathcal{L}^{r_i}})$ and for all $b \in C(\mathcal{A}_{\mathcal{L}^{r_i}})$, $\delta(b) \subseteq C(\mathcal{A}_{\mathcal{L}^{r_i}})$. This $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ implies a set of dependent rules including r_i, namely $\rho_i = \{r_j \mid I(C(\mathcal{A}_{\mathcal{L}^{r_i}}), j) \neq \emptyset\}$. We define $\Upsilon = \{\rho_i \mid r_i \in R\}$.

Example 1. Consider Σ with $n = 4$ and for all $r_1, \ldots, r_4 \in R$, we have $\mathcal{L}^{r_i} = \langle \{s_i, s_i'\}, \{a_i\}, \{\langle s_i, a_i, s_i' \rangle\}, \{s_i\} \rangle$. We also have an $\mathcal{M}$ with $\mathcal{V} = \{\langle \langle a_1, a_2, a_3, \bullet \rangle, b \rangle, \langle \langle \bullet, \bullet, \bullet, a_4 \rangle, a_4 \rangle\}$. Now, $\rho_1 = \rho_2 = \rho_3 = \{r_1, r_2, r_3\}$ and $\rho_4 = \{r_4\}$.

3. Well-formedness of Rule Sets. One condition to check property preservation is that the $\rho \in \Upsilon$ are *complete* w.r.t. synchronising behaviour. In other words, we check that each action in $C(\mathcal{A}_{\mathcal{L}^{r_i}})$ is in the left pattern action set of some $r_i \in \rho$: $C(\mathcal{A}_{\mathcal{L}^{r_i}}) \subseteq \bigcup_{r_j \in \rho} \mathcal{A}_{\mathcal{L}^{r_j}}$. If this does not hold, then some relevant behaviour is not present in any of the left patterns, making it impossible to determine property preservation based on the rule patterns alone. Often, such a situation can be fixed by including rules for relevant behaviour that actually do not transform anything, i.e. the left pattern is equal to the right pattern.

Another condition concerns the applicability of Σ on $\mathcal{M}$: if rule r_i contains behaviour in its left pattern that requires synchronisation, then r_i must be applicable on all occurrences of that behaviour in $\Pi[i]$. We call this *universal applicability* of r_i on $\Pi[i]$ (note that action a requires synchronisation iff $|\delta(a)| > 1$, and that $\mathbf{ran}(m_{r_i})$ refers to the range of m_{r_i}):

$$\forall r_i \in R, a \in \mathcal{A}_{\mathcal{L}^{r_i}}.|\delta(a)| > 1 \implies \forall s_1 \xrightarrow{a}_i s_2.\exists m_{r_i}.\{s_1, s_2\} \subseteq \mathbf{ran}(m_{r_i})$$

Example 2. Consider the Σ of Example 1 again, but now without r_2. Then, $\rho_1 = \{r_1, r_3\}$ is not complete, since $C(\mathcal{A}_{\mathcal{L}^{r_1}}) = \{a_1, a_2, a_3\}$, and $a_2 \notin \mathcal{A}_{\mathcal{L}^{r_1}} \cup \mathcal{A}_{\mathcal{L}^{r_3}}$. The same holds for ρ_3.

Example 3. Consider an $\mathcal{M}$ with $n = 2$ and $\Pi[1]$ having the structure $p_1 \xrightarrow{a} p_2 \xrightarrow{a} p_3 \xrightarrow{b} p_4$. Furthermore, $\mathcal{V}$ contains a rule $\langle \langle a, c \rangle, d \rangle$. We also have a Σ containing r_1 with $\mathcal{L}^{r_1}$ having the structure $s_1 \xrightarrow{a} s_2 \xrightarrow{b} s_3$. Now, r_1 is not universally applicable, since in $\Pi[1]$, not all occurrences of a are matched on by r_1. If instead, we had a synchronisation rule $\langle \langle a, \bullet \rangle, d \rangle$ in $\mathcal{V}$, then r_1 would be universally applicable, if b also requires no synchronisation.

4. Add Divergency Information. Transformation rules may introduce loops that after maximal hiding result in τ-loops. These may lead to a negative DSBB comparison result, since a non-diverging glue-state s in the left pattern of a rule can become diverging in the right. However, if the hidden system LTS of the input network already contains diverging behaviour, it could be the case that the matches of that rule only relate to those parts of the LTS where behaviour is already diverging, i.e. each process state p matched on by s is only part of diverging system state vectors in the system LTS. The introduction of additional

Fig. 4. An extended transformation rule

diverging behaviour will then not lead to a system LTS that is not DSBB to the original one. Such situations can be taken into account by first of all identifying which states are diverging in the hidden system LTS (this can be done in linear time with a slightly altered version of Tarjan's Strongly Connected Component detection algorithm [27]), propagating this information back to the process LTSs (a process state p is called diverging iff there exists no system state containing p that is non-diverging), and finally, adding a τ-selfloop to each s in the patterns of a rule if s is only matched on diverging process states.

Step 4 is optional, since it should only be done for transformation rules that do not *remove* diverging behaviour, since the added τ-loops will result in ignoring such removal. Removal of diverging behaviour can be detected by checking that each τ-loop in the left-pattern of a rule is represented in the right pattern.

5. Generate and Hide Relevant LTSs. To check property preservation of Σ, we need to make some structural information explicit in its rule patterns. If in a process LTS, a state is matched on by a glue-state, then it will remain in the LTS after transformation. This should be incorporated in a DSBB comparison. Consider the example in Figure 4. In this rule, the labels a and b are swapped between the transitions. Without the selfloops, a DSBB comparison will conclude that the two LTSs are equivalent, but it will not relate state ii (and iii) from the left pattern with state ii (and iii) from the right pattern, which indicates a structural change. To avoid such an erroneous conclusion, we introduce for each glue-state j a selfloop with a unique (fresh) label κ_j, and add a synchronisation rule to $\mathcal{V}$ stating that κ_j can be fired independently. Since only from state j action κ_j can be performed, both in the left and right pattern, a positive DSBB comparison outcome necessarily depends on being able to relate state j with itself. We refer with r_i^κ to rule r_i after application of the κ-modification.

Now, each $\rho \in \Upsilon$ directly defines two vectors $\overline{v}_\mathcal{L}$, $\overline{v}_\mathcal{R}$, where for $\mathcal{G} \in \{\mathcal{L}, \mathcal{R}\}$ and all $i \in 1..n$, we have $\overline{v}_\mathcal{G}[i] = \mathcal{G}^{r_i}$ if $r_i \in \rho$, and $\overline{v}_\mathcal{G}[i] = \langle \{s\}, \emptyset, \emptyset, \{s\} \rangle$ (a place-holder) otherwise. These vectors lead to two networks $\Xi^\rho_\mathcal{L} = \langle \overline{v}_\mathcal{L}, \mathcal{V} \rangle$ and $\Xi^\rho_\mathcal{R} = \langle \overline{v}_\mathcal{R}, \mathcal{V} \cup \hat{\mathcal{V}} \rangle$. The behaviour of $\Xi^\rho_\mathcal{R}$ represents the result in the system of applying the rule system to the behaviour of $\Xi^\rho_\mathcal{L}$.

Besides this pair of LTSs, we also derive LTSs for each non-empty subset of ρ, i.e. for all sets in $2^\rho \setminus \{\emptyset\}$. The subsets represent system states where some parties are able to perform synchronisation, whereas others may not be. The need to consider these states is illustrated by Example 4 described below.

6. DSBB Comparison of LTSs. For each $\rho \in \Upsilon$, we perform a DSBB comparison on all the $(\Xi^{\rho'}_\mathcal{L}, \Xi^{\rho'}_\mathcal{R})$, with $\rho' \in 2^\rho \setminus \emptyset$.

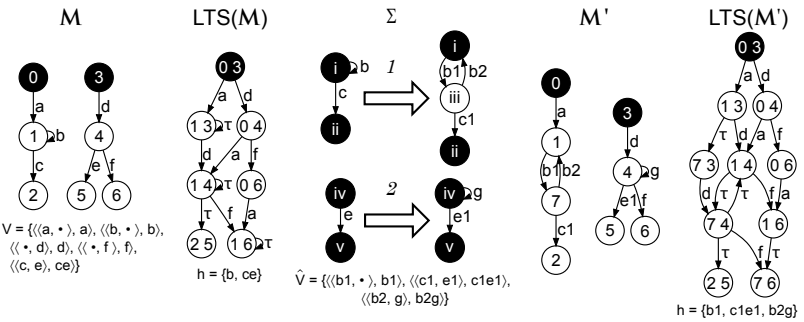

Fig. 5. The transformation of a network of LTSs

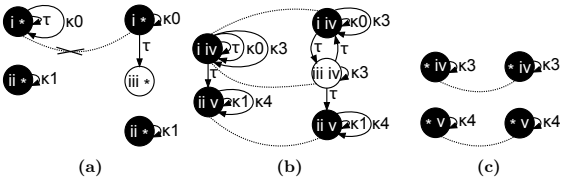

Fig. 6. DSBB comparisons of networks of (a) $\{1\}$, (b) $\{1, 2\}$, and (c) $\{2\}$

Example 4. On the left of Figure 5, a network of LTSs $\mathcal{M}$ is shown together with its system LTS after hiding $h_{\mathcal{A}_{LTS(\mathcal{M})}}(\varphi) = \{b, ce\}$, which is the hiding set for some property φ. On the right, a rule system Σ consisting of two transformation rules and some synchronisation rules to be introduced at transformation is shown. Applying Σ on $\mathcal{M}$ results in network $\mathcal{M}'$ shown to the right of Σ with its system LTS after hiding $h_{\mathcal{A}_{LTS(\mathcal{M}')}}(\varphi) = \{b1, c1e1, b2g\}$. Transformation rules 1 and 2 are clearly dependent, since actions c and e in the left patterns must synchronise according to $\mathcal{V}$. Thus, we have $\rho_1 = \rho_2 = \{1, 2\}$. In Figure 6b, the two networks of LTSs described by $\{1, 2\}$ are compared after hiding the actions in h. The dotted lines in this figure illustrate that a DSBB exists for these two networks.

Even though a DSBB exists between these networks, the system LTSs of $\mathcal{M}$ and $\mathcal{M}'$ are not DSBB. This illustrates that it is not sufficient to only look at the combination of rule patterns. Instead, we also need to consider configurations in which some parties are able to perform synchronisation, whereas some parties are not. For example, the system LTS of $\mathcal{M}$ contains a state (1 3), which has a τ-loop that cannot be simulated by the system LTS of $\mathcal{M}'$. This τ-loop is the result of hiding the b-loop of state 1 in the leftmost process of $\mathcal{M}$. This process can perform action b independently. After transformation, however, a τ-cycle can only result from synchronisation between the transformed process LTSs of $\mathcal{M}'$ ($b2$ has to synchronise with g). In system states where this required synchronisation is impossible, as is the case in the system LTS of $\mathcal{M}'$ in state (1 3), only one τ-action can be performed. By considering 'subnetworks' of $\{1, 2\}$, we detect this difference in the form of a negative DSBB comparison result (see Fig. 6a).

Correctness. Our check correctly determines whether a rulesystem is property preserving or not. This is proven in [8].

Complexity and Scalability. The bottleneck lies in computing the system LTS of a network, relating to state-space explosion. Most steps, however, do not involve this LTS. Only step 4 requires full analysis of the system LTS; however, divergency information can be propagated along property preserving transformations, i.e. it does not need to be recomputed for each new system LTS along a sequence of transformations. Given that we assume that the initial $\mathcal{M}$ can be verified, this does not introduce an additional time or space bottleneck.

Let k and m be the upper-bounds to the number of states and transitions, respectively, in a left or right rule pattern. Then, step 6 can be done for each ρ in $\mathcal{O}(2^{|\rho|} - 1 \cdot (k^{|\rho|} \cdot (k^{|\rho|} + m^{|\rho|}))$. Efficient DSBB detection takes $\mathcal{O}(k^{|\rho|} \cdot (k^{|\rho|} + m^{|\rho|}))$ [13], assuming the τ-loops have been compressed using Tarjan's algorithm, and there are $2^{|\rho|} - 1$ relevant subsets of ρ. An interesting observation is that the checks for the relevant subsets can be done fully independently, allowing for straightforward parallelisation. The space complexity of step 6 is $\mathcal{O}(k^{|\rho|} + m^{|\rho|})$.

In steps 5 and 6, worst-case, Σ would completely describe $\mathcal{M}$ in the left patterns of rules, and the right patterns would completely describe the refined model. In that case, the check actually boils down to generating the complete new LTS and checking if it is DSBB to the old one, which would not mitigate the state-space explosion problem. However, this would not be in line with the idea behind our technique. Typically, the rules in Σ contain patterns that are much smaller than the process LTSs. Then, the state spaces in step 5 will be exponentially smaller than the one of the transformed network.

Finally, our approach can only be successful if LTS transformation can be done efficiently; for a given rule, matching can be done linear to the size of the input LTS [6]. In our case, this is reasonable, as the process LTSs are usually exponentially smaller than the system LTS. Besides that, the use of transition labels means that we usually do not experience the worst-case complexity.

5 Experimental Results

Our check can be performed fully automatically by a new tool called Refiner, which integrates with the model checking toolsets CADP [10] and mCRL2 [12]; e.g., μ-calculus formulas can be verified using CADP, and in fact the mCRL2 tool *ltscompare* is used by Refiner to perform DSBB comparisons. Refiner has been implemented in Python, and can be run very efficiently using the Pypy interpreter[1]. Both Refiner and the CADP tool Exp.Open [19] can generate the system LTSs of networks; the latter is more efficient in doing so, but Refiner also stores how combinations of process states relate to the system states, which is required when computing divergency information in step 4 of our check.

We validated our approach using nine case studies on a machine with a quad-core intel xeon E5520 2.27 GHz processor, 1 TB RAM, running Fedora 12.

[1] `http://www.pypy.org`

Table 1. LTS generation results

		LTS size (# states)	time (sec.)
ACS	$\mathcal{M}_0$	3,484	1.93
	$\mathcal{M}_1$	21,936	9.95
1394-fin	$\mathcal{M}_0$	198,692	13.95
	$\mathcal{M}_1$	6,679,222	305.02
wafer	$\mathcal{M}_0$	78,919	15.29
	$\mathcal{M}_1$	474,457	96.97
broadcast	$\mathcal{M}_0$	1,024	86.77
	$\mathcal{M}_1$	60,466,176	3486.76
	$\mathcal{M}_2$	60,466,176	4259.79
ABP	$\mathcal{M}_0$	759,375	29.97
	$\mathcal{M}_1$	380,204,032	26,509.29
	$\mathcal{M}_2$	656,356,768	56,365.93
HAVi-LE	$\mathcal{M}_0$	15,688,570	587.55
	$\mathcal{M}_1$	190,208,728	7,343.60
	$\mathcal{M}_2$	3,048,589,069	335,130.67
Sieve	$\mathcal{M}_0$	6,539,813	4,003.58
	$\mathcal{M}_1$	19,434,968	12,117.29
	$\mathcal{M}_2$	135,159,971	84,893.19
ODP	$\mathcal{M}_0$	91,394	26.73
	$\mathcal{M}_1$	7,699,456	117.13
DES	$\mathcal{M}_0$	64,498,297	771.26
	$\mathcal{M}_1$	64,498,317	814.20

Table 2. Preservation checking results

		hiding (sec.)	div. (sec.)	φ-pres. #	φ-pres. (sec.)	φ
ACS	$\mathcal{M}_0 \to \mathcal{M}_1$	0.26	0.37	56	10.26	✓
1394-fin	$\mathcal{M}_0 \to \mathcal{M}_1$	0.79	3.21	36	8.30	✓
wafer	$\mathcal{M}_0 \to \mathcal{M}_1$	0.85	1.57	17	3.21	✓
broadcast	$\mathcal{M}_0 \to \mathcal{M}_1$	0.48	1.07	4	0.90	✗
	$\mathcal{M}_0 \to \mathcal{M}_2$	-	-	70	21.10	✓
ABP	$\mathcal{M}_0 \to \mathcal{M}_1$	5.46	15.74	22	5.63	✗
	$\mathcal{M}_0 \to \mathcal{M}_2$	-	-	315	19.95	✓
HAVi-LE	$\mathcal{M}_0 \to \mathcal{M}_1$	325.52	690.28	127	39.74	✓
	$\mathcal{M}_1 \to \mathcal{M}_2$	-	-	31	6.02	✓
Sieve	$\mathcal{M}_0 \to \mathcal{M}_1$	85.77	215.00	51	45.86	✓
	$\mathcal{M}_1 \to \mathcal{M}_2$	-	-	51	25.25	✓
ODP	$\mathcal{M}_0 \to \mathcal{M}_1$	1.71	3.54	31	8.30	✓
DES	$\mathcal{M}_0 \to \mathcal{M}_1$	792.69	1468.86	3	255.14	✓

For each case study, we performed a number of refinements, and both verified the resulting system LTSs and checked property preservation of the refinements. We chose not to compare with other incremental approaches (see Section 6), because the latter support only transformations of 'flat' system LTSs, while we focus on refinements of individual process LTSs in a network. In particular, other approaches do not consider the interaction of processes in a system.

Table 1 displays the size in number of states and the verification time in seconds of the relevant system LTSs using Exp.Open, where $\mathcal{M}_0$ is the initial model. The first three cases stem from the set of examples distributed with the mCRL2 toolset, the last four are slightly altered versions of CADP models, and *ABP* and *broadcast* are two cases modelled by us. For each, we applied one of three different types of refinements to the process LTSs in their network: 1) adding non-synchronising transitions, representing additional internal computations or logging of messages (the first three and the last three cases), 2) adding support for lossy channels by introducing instances of the Alternating Bit Protocol (the ABP case), and 3) breaking down broadcast synchronisations into sequences of two-party synchronisations (the broadcast and the HAVi leader election case). All three types introduce new behaviour irrelevant for the property to be checked.

Table 2 displays the numbers related to preservation checking: per transformation, the time needed to hide irrelevant actions, the time needed to compute divergencies, and information related to the actual checking is shown. For the checking, the number of DSBB comparison checks, the total runtime, and the outcome of the check is displayed. The time needed for transformation is not displayed, but it takes at most as long as preservation checking, since the latter also involves rule matching. Note that hiding and divergency computation is only required before applying the first transformation on the initial model; for the same property, subsequent transformations can reuse the hidden network, and divergency information is updated when transforming. For the refinements of type 1, the standard preservation check sufficed, but for the other refinements,

divergency information was required. The results clearly show the benefits of our approach: when the check concludes that a rule system preserves a property, exploration of the resulting system LTS can be avoided; this is fruitful if the transformation does not alter the LTS that much, as is the case for the DES protocol,[2] but the check really pays off when transformation leads to much larger LTSs. For example, in the HAVi leader election case, we have one subnetwork of three managers and one of three messaging systems, both of which involve three-party synchronisation. One practical refinement is to break these down into several two-party synchronisations, and in two transformation steps, this leads to models $\mathcal{M}_1$ and $\mathcal{M}_2$. Completely analysing the system LTS of $\mathcal{M}_2$ takes 93 hours, but the check can be done in about 6 seconds if hiding and divergency computation has already been done, and 17 minutes if this is not the case. In this case, we hid all behaviour irrelevant for a particular liveness property.

6 Related Work

Our work is related to incremental model checking. Early papers on this subject propose techniques to reuse model checking results of safety properties for a given LTS to determine whether it still satisfies the same property after some alterations [25,26]. Large speedups are reported compared to complete rechecking, but the memory requirements are at least as high, since all states plus additional bookkeeping per state must reside in memory. Our technique does not require this. Furthermore, we do not deal with large, flat LTSs directly, but with networks and transformation rules that both consist of relatively small LTSs. Finally, we do not recheck a property after transformation, but check bisimulation instead.

In the context of *Dynamic graph algorithms* [9], reachability is an *unbounded* problem [23,25], i.e. it cannot be determined solely based on the changes. Thanks to the gluing conditions and our criteria, this is not an issue in our context.

Saha [24] presents an incremental algorithm for updating bisimulation relations based on changes of a graph. The goal of Saha is to efficiently maintain a bisimulation, whereas the goal of our work is to assess whether a bisimulation is guaranteed to remain without actually constructing or maintaining it.

Work on finding refinement mappings, e.g. [1], is related, but the question whether there exists a mapping between two given models, establishing that one is an implementation of another, is different from having a model and a formalisation of how to transform it, and asking whether the transformation will preserve a property without looking at the application result. Work related to B, e.g. [20], is on strictly refining existing functionalities. We also support adding new functionality, as long as it is not relevant for the desired property.

Monotonically adding functionality, as opposed to refining, is addressed in e.g. [3]. The focus is on updating property formulae; it could be interesting to see if this is applicable in our setting to update properties.

[2] Note the long runtime of the divergency computation for the DES protocol, relative to generating its LTS with Exp.Open. Further improvement of the implementation of Refiner is expected to resolve this.

Combemale et al. [5], Hülsbusch et al. [15], and Karsai and Narayanan [16,22] check semantics preservation of model transformations using either strong or weak bisimilarity. They consider transformation to other modelling languages, whereas we focus on model refinement.

7 Conclusions and Future Work

We presented a technique aimed at verifying the correctness of complex models that are the result of iterative refinement through model transformation. It checks whether safety, liveness, and fairness properties are preserved by rule systems if they are well-formed w.r.t. the semantics of the input model. If a rule system preserves a property that holds for a given input model, construction and exploration of the new LTS can be avoided. Experiments show that preservation checking is several orders of magnitude faster than rechecking the property.

For future work, first, the concept of networks of LTSs could be extended to support additional features such as asynchronous communication. Furthermore, the relation between the formal notion of rule system and practical languages for the implementation of model transformations needs further study.

References

1. Abadi, M., Lamport, L.: The Existence of Refinement Mappings. Theoretical Computer Science 82, 253–284 (1991)
2. Beydeda, S., Book, M., Gruhn, V. (eds.): Model-Driven Software Development. Springer, Heidelberg (2005)
3. Braunstein, C., Encrenaz, E.: CTL-Property Transformations Along an Incremental Design Process. In: Proceedings of the Fourth International Workshop on Automated Verification of Critical Systems. Electronic Notes in Theoretical Computer Science, vol. 128, pp. 263–278. Elsevier (2004)
4. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
5. Combemale, B., Crégut, X., Garoche, P.-L., Thirioux, X.: Essay On Semantics Definition in MDE - An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (2009)
6. Dodds, M., Plump, D.: Graph Transformation in Constant Time. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 367–382. Springer, Heidelberg (2006)
7. Ehrig, H., Pfender, M., Schneider, H.: Graph Grammars: an Algebraic Approach. In: IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory, pp. 167–180. IEEE (1973)
8. Engelen, L.J.P., Wijs, A.J.: Checking Property Preservation of Refining Transformations for Model-Driven Development. CS-Report 12-08, Eindhoven University of Technology (2012)
9. Eppstein, D., Galil, Z., Italiano, G.: Dynamic Graph Algorithms. In: CRC Handbook of Algorithms and Theory of Computation, ch. 22. CRC Press (1997)
10. 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)

11. van Glabbeek, R.J., Luttik, B., Trčka, N.: Branching Bisimilarity with Explicit Divergence. Fundamenta Informaticae 93(4), 371–392 (2009)
12. Groote, J.F., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., van Weerdenburg, M., Wesselink, W., Willemse, T., van der Wulp, J.: The mCRL2 Toolset. In: Proceedings of the 1st International Workshop on Academic Software Development Tools and Techniques (2008)
13. Groote, J.F., Vaandrager, F.: An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 626–638. Springer, Heidelberg (1990)
14. Heckel, R.: Graph Transformation in a Nutshell. In: Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques. Electronic Notes in Theoretical Computer Science, vol. 148, pp. 187–198. Elsevier (2006)
15. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., 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. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop 2006. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2007)
17. Kozen, D.: Results on the Propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
18. Lambers, L., Ehrig, H., Orejas, F.: Efficient Detection of Conflicts in Graph-based Model Transformation. In: Proceedings of the International Workshop on Graph and Model Transformation. Electronic Notes in Theoretical Computer Science, vol. 152, pp. 97–109. Elsevier (2006)
19. Lang, F.: Exp.Open 2.0: A Flexible Tool Integrating Partial Order, Compositional, and On-The-Fly Verification Methods. In: Romijn, J., Smith, G., van de Pol, J. (eds.) IFM 2005. LNCS, vol. 3771, pp. 70–88. Springer, Heidelberg (2005)
20. Lano, K.: The B Language and Method, A Guide to Practical Formal Development. Springer, Heidelberg (1996)
21. Mateescu, R., Wijs, A.: Property-Dependent Reductions for the Modal Mu-Calculus. In: Groce, A., Musuvathi, M. (eds.) SPIN 2011. LNCS, vol. 6823, pp. 2–19. Springer, Heidelberg (2011)
22. Narayanan, A., Karsai, G.: Towards Verifying Model Transformations. In: Proceedings of the International Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Notes in Theoretical Computer Science, vol. 211, pp. 191–200 (2008)
23. Ramalingam, G., Reps, T.: On The Computational Complexity of Dynamic Graph Problems. Theoretical Computer Science 158, 233–277 (1996)
24. Saha, D.: An Incremental Bisimulation Algorithm. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 204–215. Springer, Heidelberg (2007)
25. 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)
26. Swamy, G.M.: Incremental Methods for Formal Verification and Logic Synthesis. PhD thesis, University of California (1996)
27. Tarjan, R.: Depth-First Search and Linear Graph Algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)