Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

# Evolution of Model Transformations by Model Refactoring: Long Version

Hartmut Ehrig, Karsten Ehrig and Claudia Ermel

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{ehrig,karstene,lieske}@cs.tu-berlin.de

Bericht-Nr. 2009/04, ISSN 1436-9915

## Abstract

Model-to-model transformations between visual languages are often defined by typed, attributed graph transformation systems. Here, the source and target languages of the model transformation are given by type graphs (or meta models), and the relation between source and target model elements is captured by graph transformation rules. On the other hand, refactoring is a technique to improve the structure of a model in order to make it easier to comprehend, more maintainable and amenable to change. Refactoring can be defined by graph transformation rules, too. In the context of model transformation, problems arise when models of the source language of a model transformation become subject to refactoring. It may well be the case that after the refactoring, the model transformation rules are no longer applicable because the refactoring induced structural changes in the models. In this paper, we consider a graph-transformation-based evolution of model transformations which adapts the model transformation rules to the refactored models. In the main result, we show that under suitable assumptions, the evolution leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to a well-known model transformation from statecharts to Petri nets.

## 1 Introduction

Model-driven software development (MDD) is a discipline that relies on models and that aims to develop, maintain and evolve software by performing model transformations [1]. The basic idea of model transformations is to more or less automatically derive models of a certain target language from models of a source language, e.g. by mapping the source language components of a domain specific language to Petri nets, where model properties can be analyzed formally.

An intrinsic property of software (and their models) in a real-world environment is their need to evolve. As the model is enhanced, modified and adapted to new requirements, it becomes more and more complex and drifts away from its original design. *Refactoring* [2, 3], originally used in the industry for source code re-structuring, aims at reducing the software complexity by "changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [2]. Recently, approaches for refactoring have been lifted to the more abstract level of design models (*model refactoring*), supporting in particular the refactoring of UML diagrams like class diagrams, statecharts and activity diagrams [4, 5].

In this paper we tackle the problem which arises when model refactoring operations are applied to a model (or a modelling language) which is transformed by a model transformation. Problems arise if the refactoring operations induce structural model changes which cannot be handled by the model transformation. In order to solve this problem, we propose a strategy for a systematic evolution of model transformation specifications in accordance to the refactoring operations.

Model transformations between visual languages are conveniently defined in a formal way by typed, attributed graph transformation [6–9]. To execute model transformation rules and to check functional properties of model transformations (termination and confluence), the graph transformation engine AGG [10] is available.

On the other hand, various approaches exist using graph transformation to provide a formal specification of model refactorings [11–13]. Basically, a refactoring operation is defined by a set of graph transformation rules typed over the modelling language of the models to be refactored.

In our approach, we consider a construction allowing us to apply the refactoring operation not only to models of the source or target language of a model transformation, but also to the model transformation rules. The approach is based on the work of Parisi-Presicce who defined the transformation of graph grammars in [14]. In our main result, we show that under suitable assumptions, such an evolution of the model transformation rules leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to a well-known model transformation from statecharts to Petri nets, when the statechart becomes subject to a refactoring.

This technical report is the long version of our contribution to the Workshop of Graph Transformation and Visual Modeling Techniques 2009 [15].

The paper is structured as follows: After introducing our case study for refactoring and model transformation in Section 2, we consider the notion of consistency of a model transformation step and a refactoring step in Section 3, where the steps are defined as single rule applications of the respective graph rules to a model state. In Section 4, we extend this basis to sequences of rule applications and state our main result for the consistent evolution of model transformations. We give an overview over extension of our main results in Section 5, and look into some further refactorings in Section 6. Section 7 compares our approach to related work, and in Section 8 we conclude the paper with an outlook to future work.

## 2 Example: Transforming and Refactoring Statecharts

### 2.1 Model Transformation *State2PN* from Statecharts to Petri Nets

In this section, we review the model transformation from a simple version of statecharts into Petri nets, given in [6].

*Example 1 (Type Graph of the SC2PN Model Transformation).* The statechart type graph $TG_S$ is shown in the left part of Fig. 1 and explicitly introduces several ideas from the area of statecharts that are only implicitly present in the standard UML metamodel (such as state configurations). We consider a network of state machines StateMachine. A single state machine captures the behavior of any object of a specific class by flattening the state hierarchy into *state configurations* and grouping parallel transitions into *steps*. A Configuration is composed of a set of States that can be active at the same time. A Step is composed of non-conflicting Transitions (which are, in turn, binary relations between states) that can be fired in parallel. A step between two configurations is triggered by a common Event for all its transitions. The effect of a step is a set of Actions.

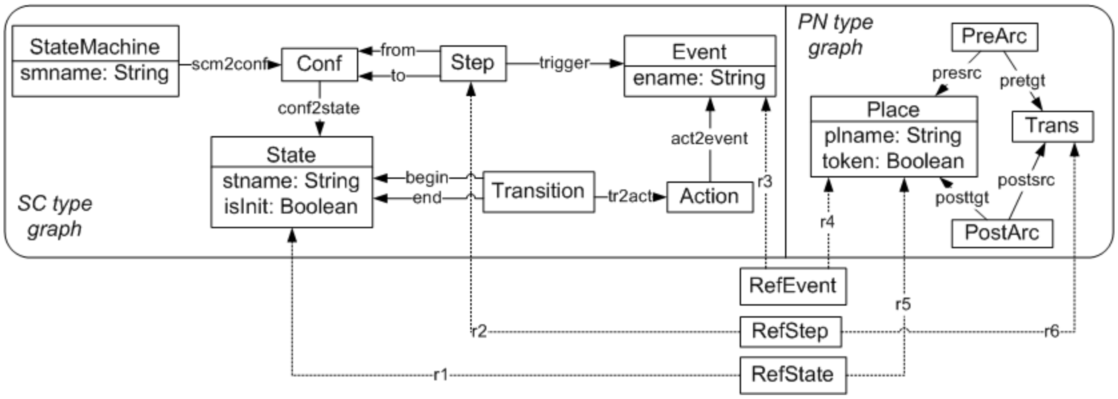


**Fig. 1.** Integration of Attributed Type Graphs for the Model Transformation $SC2PN$

The target modelling language are Petri nets. The Petri net type graph $TG_T$ is shown in the right part of Fig. 1. In fact, we use elementary net systems [16], where each place contains at most one token. In order to interrelate the source and target modeling languages, we use reference types to construct an integrated attributed type graph, as shown in Fig. 1. For instance, the reference node type RefState relates the source type State to the target type Place.

The model transformation from statecharts into Petri nets is fully given by the transformation rules defined in [6]. In this paper, we concentrate on the rules constructing the integrated model which contains elements of both source and target language, and do not consider explicitly the restriction of the integrated model to the target language of Petri nets.

The main model transformation rules are shown in Fig. 2. Note that we use a shortcut notation for our rules where the left- and right-hand sides of each rule are depicted in one graph. Nodes which exist only in the right-hand side (i.e. they are generated by the rule) are coloured, and their adjacent arcs are also generated by the rule. Moreover, all model transformation rules are *non-deleting*, and each rule has a negative application condition (NAC) which equals the right-hand rule side and prevents the rule to be applied more than once at the same match as before.

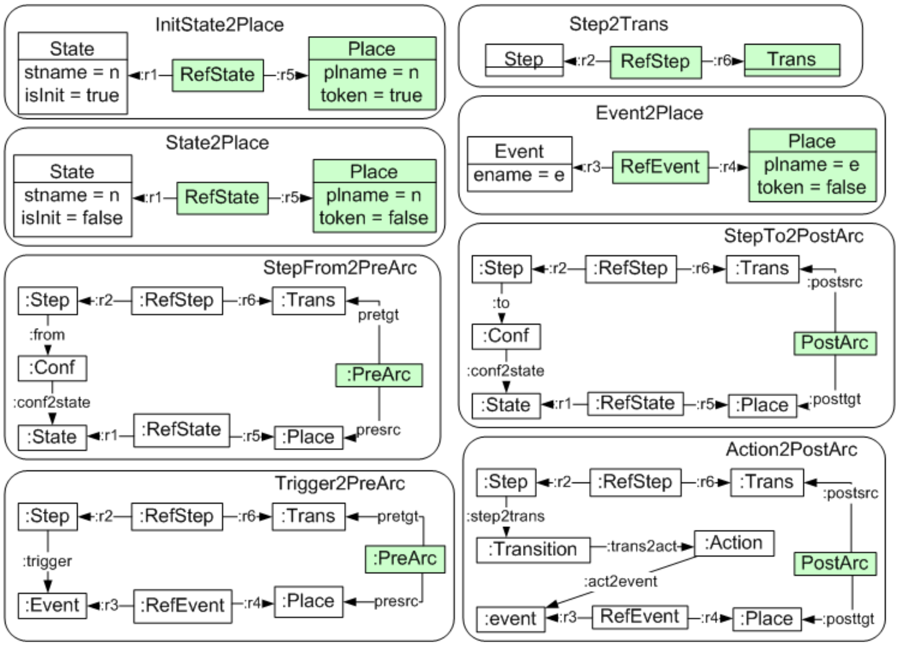
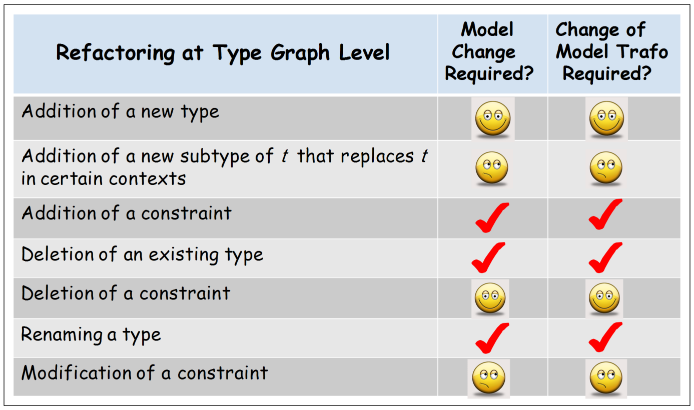


**Fig. 2.** Model Transformation Rules for *SC2PN*

*Example 2 (SC2PN Model Transformation Rules).* Each state in the statechart is transformed to a corresponding place in the target Petri net model, where a token in such a place denotes that the corresponding state is active initially (rules InitState2Place and State2Place). A separate place is generated for each valid event in rule Event2Place. Each step in the statechart is transformed into a Petri net transition (rule Step2Trans). Since the Petri net should simulate how to exit and enter the corresponding states in the statechart, input and output arcs of the transition have to be generated accordingly (see rules StepFrom2PreArc and StepTo2PostArc). Furthermore, firing a transition should consume the token of the trigger event (rule Trigger2PreArc), and should generate tokens on (the places related to) the target event indicated as the action (Action2PostArc).

### 2.2 Refactoring Operation for Statecharts

Not all possible model refactorings make it necessary to adapt the model transformation rules. One well-known refactoring is the so-called *Pull-Up-Attribute* which removes an attribute type from all subtypes of a supertype and adds the attribute type to the common supertype, instead. This kind of refactoring (changing only the inheritance relation of a meta model) does not induce changes on the instance models which remain valid as they are. Hence, model transformation rules remain applicable after the refactoring, too. On the other hand, there are refactorings which induce structural changes of the instance models. This kind of critical refactorings make an adaption of the model transformation rules necessary and are considered here. Fig. 3 shows an overview of changes in the type graph and the necessity of changing (migrating) the corresponding models and/or model transformation rules, as well.



| Refactoring at Type Graph Level | Model Change Required? | Change of Model Trafo Required? |
|---|---|---|
| Addition of a new type | 🙂 | 🙂 |
| Addition of a new subtype of *t* that replaces *t* in certain contexts | 😕 | 😕 |
| Addition of a constraint | ✓ | ✓ |
| Deletion of an existing type | ✓ | ✓ |
| Deletion of a constraint | 🙂 | 🙂 |
| Renaming a type | ✓ | ✓ |
| Modification of a constraint | 😕 | 😕 |

**Fig. 3.** Relation between Refactorings at Meta-Model and at Model Level

Adding new types or deleting constraints are uncritical since existing models remain valid with respect to the new type graph, as well. Critical refactoring operations are the addition of constraints, and the deletion of existing types (including attribute types). Here, the added constraints may be violated by existing models, and deleted types may be used in existing models, which must be refactored accordingly. If the intention of adding a new subtype is that certain model elements, previously typed over the supertype, should now be typed over the new subtype, then the models must be adapted, as well. Analogously, existing models might use types which have been renamed or violate constraints after they have been modified, depending on the character of the modification.

As running example, we present a refactoring operation for statecharts, where the representation of initial states is changed from an attribute to a new node type. This involves the deletion of an attribute type which is a critical refactoring according to Fig. 3. The motivation for this statechart refactoring is to simplify the definition of a concrete syntax for statecharts, where node types are mapped to figures. We use this example later on to illustrate the evolution of a model transformation from statecharts to Petri nets when such a model refactoring on statecharts has taken place.

*Example 3 (Refactoring Operation for Statecharts).* Let the type graph for statecharts be the one depicted in the left part of Fig. 1. For the definition of our refactoring operation, this type graph is extended by two new node types Initial and Normal, which are linked to the State node type. The refactoring operation markState is modelled by the two graph rules in Fig. 4, where an Initial node is added to a state whose isInit attribute is true (rule markInitial), and, vice versa, a Normal node is added to a state whose isInit attribute is false (rule markNormal). Note that the isInit attribute is deleted by the refactoring rules.

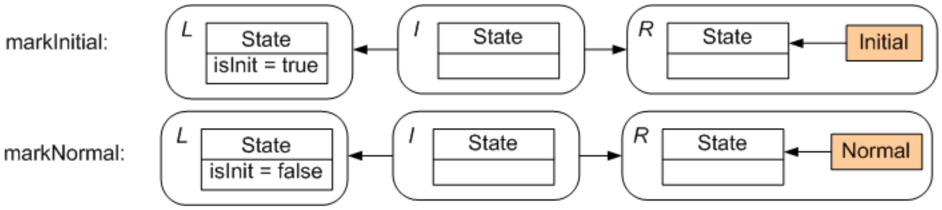


**Fig. 4.** Rules for Statechart Refactoring Operation markState

# 3 Consistency of Stepwise Model Transformation and Refactoring

In this section, we give the formal definition how to adapt a model transformation to a refactoring operation (Def. 1) and consider the relation of a model transformation step and a refactoring step in Lemma 1.

A model transformation rule $p_1 \in P$ is adapted to a refactoring (given by refactoring rule $q \in Q$), by applying refactoring rule $q$ to all rule graphs of model transformation rule $p_1$, resulting in the adapted model transformation rule $p_2$. Note that the construction of applying rules to rules is based on [14] and extended to rules with NACs in [17].

**Definition 1 (Application of $Q$-Productions to $P$-Productions).**

*Production $q = (L_q \leftarrow I_q \rightarrow R_q)$ is applicable to $p_1 : L_1 \rightarrow R_1$ with $nac_1 : L_1 \rightarrow N_1$ leading to $p_2 : L_2 \rightarrow R_2$ with $nac_2 : L_2 \rightarrow R_2$ if we have $m : L_q \rightarrow L_1$ leading to the following DPOs, written $p_1 \overset{q,m}{\Longrightarrow} p_2$ , where all morphisms are injective:*

$$\begin{tikzcd}
L_q \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I_q \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & R_q \arrow[d] \\
L_1 & D \arrow[l] \arrow[r] & L_2
\end{tikzcd}
\qquad
\begin{tikzcd}
L_1 \arrow[d, "p_1"'] \arrow[dr, phantom, "(3)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(4)"] & L_2 \arrow[d, "p_2"] \\
R_1 & E \arrow[l] \arrow[r] & R_2
\end{tikzcd}
\qquad
\begin{tikzcd}
L_1 \arrow[d, "nac_1"'] \arrow[dr, phantom, "(5)"] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(6)"] & L_2 \arrow[d, "nac_2"] \\
N_1 & F \arrow[l] \arrow[r] & N_2
\end{tikzcd}$$

*Example 4 (Applying a Refactoring Rule to a Model Transformation Rule).* Fig. 5 shows the application of refactoring rule markInitial from Fig. 4 to model transformation rule InitState2Place from Fig. 2, according to Def. 1.

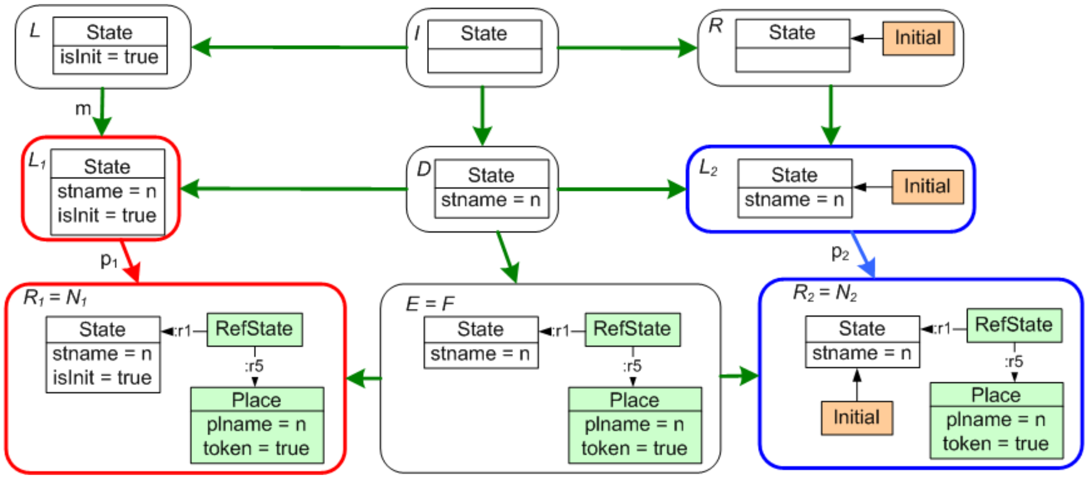


**Fig. 5.** Applying Refactoring Rule *markInitial* to Model Transformation Rule *InitState2Place*

*General Assumption:* Let a visual modeling language $VL$ be given by all models (graphs) typed over a type graph. As basis for model transformation and refactoring, we assume a common type graph $TG$ which includes the type graphs for the source and the target languages of the model transformation, as well as the extended type graph for the refactoring. Let $(MT, P) : VL_1 \rightarrow VL_2$ be a model transformation (with $P$ non-deleting with NACs), $(MR_1, Q) : VL_1 \rightarrow VL_1^*$ be a model refactoring (with $Q$ bijective on nodes, without NACs), and $(MR, Q) : P \rightarrow P^*$ be a model refactoring of rules according to Def. 1, and let $TG$ be the common type graph for $VL_1, VL_2, VL_1^*, P$ and $Q$. All over, we assume injective rules and injective matches. For simplicity, we do not handle the corresponding refactorings of the different type graphs in this paper.

The following lemma shows the compatibility of a model transformation step transforming source model $G_1 \in VL_1$ into target model $G_2 \in VL_2$ by applying rule $p_1 \in P$, and a refactoring step, changing $G_1 \in VL_1$ to $G_1' \in VL_1^*$ by applying rule $q \in Q$, where the refactored source model $G_1'$ is transformed by the refactored model transformation rule $p_2 \in P^*$, resulting in model $G_2'$.

**Lemma 1 (Direct Transformation and Refactoring Steps).**

*Given $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ with $p_1 \in P$ and $p_1 \overset{q,m}{\Longrightarrow} p_2$ with $q \in Q$, we have $G_1 \overset{q}{\Longrightarrow} G_1', G_2 \overset{q}{\Longrightarrow} G_2'$ and $G_1' \overset{p_2}{\Longrightarrow} G_2'$.*

$$\begin{tikzcd}
G_1 \arrow[rr, Rightarrow, "p_1{,}m_1"] \arrow[d, Rightarrow, "q"'] & {} \arrow[d, Rightarrow, "q{,}m"] & G_2 \arrow[d, Rightarrow, "q"] \\
G_1' \arrow[rr, Rightarrow, "p_2"'] & {} & G_2'
\end{tikzcd}$$

*Proof.* Given $p_1 : L_1 \to R_1$ with $nac_1 : L_1 \to N_1$, we obtain $p_2 : L_2 \to R_2$ with $nac_2 : L_2 \to N_2$ with pushouts $(1) - (6)$ as in Def. 1.

Furthermore, we obtain from $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ the pushout in the left square in the diagram below, with pushouts $(1) - (4)$, as shown in Def. 1. Next, we construct $D_1$ as pushout complement in the left back square – using that $I_q \to L_q$ and hence $D \to L_1$ is bijective on nodes, which implies that the gluing condition is satisfied – and then $G_1'$ as pushout in the right back square. Then, $D_2$ and $G_2'$ are constructed as pushouts in the middle and right square, respectively, leading to induced morphisms $D_2 \to G_2$ and $D_2 \to G_2'$ such that all squares commute.

$$\begin{tikzcd}
 & L_q \arrow[d, "m"'] \arrow[drr, phantom, "(1)"] & & I_q \arrow[ll] \arrow[rr] \arrow[d] \arrow[drr, phantom, "(2)"] & & R_q \arrow[d] \\
 & L_1 \arrow[dl, "p_1"'] \arrow[dd, "m_1"] \arrow[dr, phantom, "(3)"] & & D \arrow[ll] \arrow[rr] \arrow[dl] \arrow[dd] \arrow[dr, phantom, "(4)"] & & L_2 \arrow[dl, "p_2"] \arrow[dd] \\
R_1 \arrow[dd] & & E \arrow[ll] \arrow[rr] \arrow[dd] & & R_2 \arrow[dd] & \\
 & G_1 \arrow[dl] & & D_1 \arrow[ll] \arrow[rr] \arrow[dl] & & G_1' \arrow[dl] \\
G_2 & & D_2 \arrow[ll, dotted] \arrow[rr, dotted] & & G_2' &
\end{tikzcd}$$

In the left cube, the left, right, back and top squares are pushouts by construction. This implies that also the front and bottom squares are pushouts by pushout composition and decomposition. Hence, all squares of the left cube and, similarly, also of the right cube are pushouts. This leads to the DPOs of the direct transformations $G_1 \overset{q}{\Longrightarrow} G_1'$, $G_2 \overset{q}{\Longrightarrow} G_2'$ and $G_1' \overset{p_2,m_2}{\Longrightarrow} G_2'$.

It remains to show that $m_2 : L_2 \to G_1'$ satisfies $nac_2 : L_2 \to N_2$, defined by pushouts (5) and (6) in Def. 1, using that $m_1 : L_1 \to G_1$ satisfies $nac_1 : L_1 \to N_1$. Assume that $m_2 \not\models nac_2$, then we have injective $q_2 : N_2 \to G_1'$ with $m_2 = q_2 \circ nac_2$. Pushout-pullback decomposition allows us to construct pushouts (7) and (8) from the outer DPO, leading to an injective $q_1$ with $q_1 \circ nac_1 = m_1$. This contradicts $m_1 \models nac_1$. Hence, we have $m_2 \models nac_2$.

$$\begin{tikzcd}
L_1 \arrow[d, "nac_1"'] \arrow[dr, phantom, "(5)"] \arrow[dd, bend right=60, "m_1"'] & D \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(6)"] & L_2 \arrow[d, "nac_2"] \arrow[dd, bend left=60, "m_2"] \\
N_1 \arrow[d, "q_1"'] \arrow[dr, phantom, "(7)"] & F \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(8)"] & N_2 \arrow[d, "q_2"] \\
G_1 & D_1 \arrow[l] \arrow[r] & G_2
\end{tikzcd}$$

*Example 5 (Model Transformation Step and Refactoring Step).*

Fig. 6 shows the diagram relating the source and target model of the model transformation step and the changed source and target models of the refactoring step where $p_1$ and $p_2$ are given in Fig. 5.

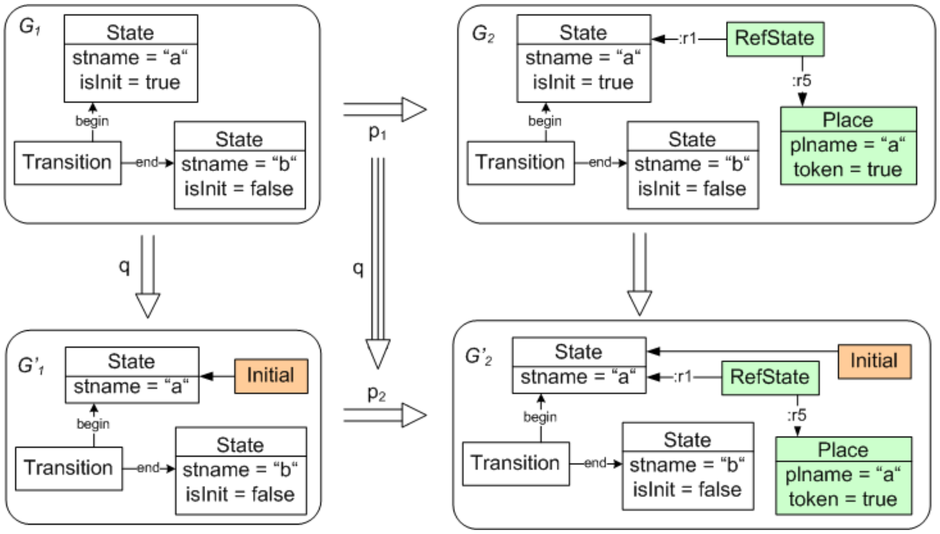


**Fig. 6.** Relating Refactoring and Model Transformation Step

## 4 Sequences of Rule Applications

In this section, we extend our result from Lemma 1 on the compatibility of model transformation and refactoring steps to sequences with rule sets $Q$, $P$ and $P^*$ according to the general assumption in Section 3. Our main result in Thm. 1 states that under certain compatibility assumptions which can be decided at rule level, a complete model transformation sequence can be refactored, leading to a compatibility diagram similar to the one in Lemma 1, but where now sequences of rule applications are considered instead of single steps. For the proof of Thm. 1, we require compatibility of model transformation and refactoring rules, defined in Def. 2. Furthermore, we use a lemma stating that a terminating transformation at rule level leads to a terminating transformation at model level, as well (Lemma 2). We say that graph $G$ (resp. rule $p^*$) is terminal wrt. $Q$ if no rule $q \in Q$ can be applied to $G$ (resp. $p^*$).

**Definition 2 ($Q$– ($P$, $P^*$)– Compatibility).**
*$Q$ is $(P, P^*)$-compatible if we have:*

1. *Independence Compatibility:*
   *Given terminal $p^*$ wrt. $Q$, $G_1 \overset{p^*}{\Longrightarrow} G_2$ and $G_1 \overset{q}{\Longrightarrow} G'_1$ (resp. $G_2 \overset{q}{\Longrightarrow} G'_2$) with $p^* \in P^*$ and $q \in Q$, we have parallel (resp. sequential) independence including NACs of $G_2 \overset{p^*}{\Longleftarrow} G_1 \overset{q}{\Longrightarrow} G'_1$ (resp. $G_1 \overset{p^*}{\Longrightarrow} G_2 \overset{q}{\Longrightarrow} G'_2$ for terminal $G_1$ wrt. $Q$).*
2. *Termination Compatibility:*
   *For each $G$ terminal wrt. $P$ and $G \overset{Q!}{\Longrightarrow} G^*$, also $G^*$ is terminal wrt. $P^*$, where $Q!$ means to apply rules in $Q$ as long as possible.*

*Example 6 (Compatibility of the SC2PN Model Transformation and the markState Refactoring).*

We continue our case study introduced in Examples 1 - 5. Fig. 7 shows the refactored model transformation rules InitState2Place and State2Place. Note that all other model transformation rules from Fig. 2 remain unchanged because the refactoring rules cannot be applied to them.

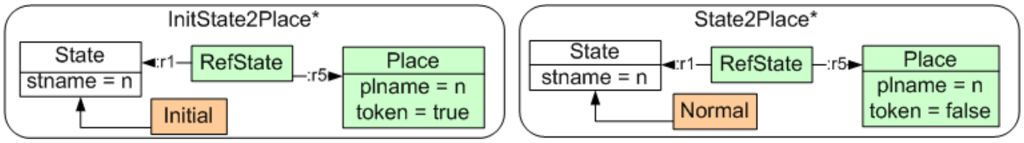


**Fig. 7.** Refactored Model Transformation Rules for *SC2PN*

We now show that we have independence and termination compatibility as defined in Def. 2:

1. Independence compatibility: Given terminal $p^*$ wrt. $Q$ and $q \in Q$ with $G_1' \overset{q}{\Longleftarrow} G_1 \overset{p^*}{\Longrightarrow} G_2$, we have parallel independence because the matches can only overlap in State which is a gluing point for both rules. Moreover, we have NAC compatibility because the nodes and edges generated by the rules in $Q$ are of different types from those generated by $p^*$. Analogously, we can show sequential independence.
2. Termination compatibility: Given terminal $G$ wrt. $P$ and $G \overset{Q!}{\Longrightarrow} G^*$, then the *markState* refactoring rules have been applied to all initial state nodes occuring in a rule in $P$, and to all initial state nodes in $G$. So there is no match from a rule $p^* \in P^*$ to $G^*$ where the NAC of $p^*$ would not prevent its application, and hence, $G^*$ is terminal wrt. $P^*$.

The following lemma states that a terminating transformation at rule level leads to a terminating transformation at model level.

**Lemma 2 (Direct Transformation and Terminating Refactoring).**

*Given $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ with $p_1 \in P$ and $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ terminating, we construct $G_1^* \overset{p_1^*}{\Longrightarrow} G_2^*$ and terminating $G_1 \overset{Q!}{\Longrightarrow} G_1*$ and $G_2 \overset{Q!}{\Longrightarrow} G_2^*$, provided that we have termination of $(MR_1, Q)$ and independence compatibility (see Def. 2.1).*

*Proof.* Let $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ terminate via $(q_1, .., q_n)$ and $G_1 \overset{p_1}{\Longrightarrow} G_2$, then we apply Lemma 1 in each step, leading to diagrams $(1) - (n)$.

$$\begin{tikzcd}
G_1 \arrow[r, Rightarrow, "q_1"] \arrow[d, Rightarrow, "p_1"'] \arrow[dr, phantom, "(1)"] & G_{11} \arrow[r, Rightarrow, "q_2"] \arrow[d, Rightarrow, "p_{11}"] \arrow[dr, phantom, "(2)"] & G_{12} \arrow[r, dotted, no head] \arrow[d, Rightarrow, "p_{12}"] & {} \arrow[d, Rightarrow, "p_{1_{n-1}}"'] \arrow[r, Rightarrow, "q_n"] \arrow[dr, phantom, "(n)"] & G_{1n} \arrow[r, Rightarrow, "q_{n+1}"] \arrow[d, Rightarrow, "p_{1n}"', "p_1^*"] \arrow[dr, phantom, "(n+1)"] & G_{1_{n+1}} \arrow[r, dotted, no head] \arrow[d, Rightarrow, "p_1^*"] & {} \arrow[r, Rightarrow, "q_{n+m}"] \arrow[d, Rightarrow, "p_1^*"'] \arrow[dr, phantom, "(n+m)"] & G_1^* \arrow[d, Rightarrow, "p_1^*"] \\
G_2 \arrow[r, Rightarrow, "q_1"'] & G_{21} \arrow[r, Rightarrow, "q_2"'] & G_{22} \arrow[r, dotted, no head] & {} \arrow[r, Rightarrow, "q_n"'] & G_{2n} \arrow[r, Rightarrow, "q_{n+1}"'] & G_{2_{n+1}} \arrow[r, dotted, no head] & {} \arrow[r, Rightarrow, "q_{n+m}"'] & G_2^*
\end{tikzcd}$$

If $G_{1_n}$ is not yet terminal wrt. $Q$, we can extend $G_1 \overset{*}{\Longrightarrow} G_{1_n}$ by $G_{1_n} \overset{Q!}{\Longrightarrow} G_1^*$ via $(q_{n+1},..,q_{n+m})$ with terminal $G_1^*$ wrt. $Q$, using termination of $(MR_1, Q)$. Parallel independence of $G_{1_n} \overset{p_1^*}{\Longrightarrow} G_{2_n} \overset{q_{n+1}}{\Longrightarrow} G_{1_{n+1}}$ according to independence compatibility allows us to construct diagram $(n+1)$ by the Local Church-Rosser Theorem with NACs, and, similarly, diagrams $(n+2),..,(n+m)$. But now also $G_2 \overset{*}{\Longrightarrow} G_2^*$ via $(q_1,..,q_{n+m})$ is terminating because $G_2^* \overset{q}{\Longrightarrow} G_2^{**}$ would imply $G_1^* \overset{q}{\Longrightarrow} G_1^{**}$ by sequential independence of $G_1^* \overset{p_1^*}{\Longrightarrow} G_2^* \overset{q}{\Longrightarrow} G_2^{**}$ according to independence compatibility.

Now we state our main result saying that under certain compatibility assumptions which can be decided at rule level, a complete model transformation sequence can be refactored, leading to a compatibility diagram similar to the one in Lemma 1, but where now sequences of rule applications are considered instead of single steps.

**Theorem 1 (Evolution of Model Transformations by Model Refactoring).** *Given a model transformation $(MT, P) : VL_1 \to VL_2$ (with $P$ nondeleting with NACs), a model refactoring $(MR_1, Q) : VL_1 \to VL_1^*$ (with $Q$ bijective on nodes, without NACs), and a model refactoring $(MR, Q) : P \to P^*$ according to Def. 1 with common type graph $TG$ for $VL_1, VL_2, VL_1^*, P$ and $Q$, such that*

1. *$(MT, P), (MR_1, Q)$ and $(MR, Q)$ are terminating,*
2. *$Q$ is locally confluent,*
3. *$Q$ is $(P, P^*)$-compatible (see Def. 2),*

*then we have $VL_2^*$ typed over $TG$ with extended*

4. *terminating model refactoring $(MR_2, Q) : VL_2 \to VL_2^*$, and*
5. *terminating model transformation $(MT^*, P^*) : VL_1^* \to VL_2^*$ with*
6. *commutativity of the diagram to the right.*

$$\begin{tikzcd}
VL_1 \arrow[r, "(MT{,}P)"] \arrow[d, "(MR_1{,}Q)"'] & VL_2 \arrow[d, "(MR_2{,}Q)"] \\
VL_1^* \arrow[r, "(MT^*{,}P^*)"'] & VL_2^*
\end{tikzcd}$$

*Proof.* Given $G_1 \in VL_1$, $G_1 \overset{Q!}{\Longrightarrow} G_1^*$, $G_1 \overset{P!}{\Longrightarrow} G_2$ via $(p_1,..,p_n)$, and $p_i \overset{Q!}{\Longrightarrow} p_i^*$ for $(i = 1,..,n)$, where termination is given by assumption 1. Now, we use Lemma 2 above to construct the following sequence $(1) - (n)$:

$$\begin{tikzcd}
G_1 \arrow[r, Rightarrow, "p_1"] \arrow[d, Rightarrow, "Q!"'] \arrow[dr, phantom, "(1)"] & G_{11} \arrow[r, Rightarrow, "p_2"] \arrow[d, Rightarrow, bend right, "Q!"] \arrow[d, Rightarrow, bend left, "Q!"'] \arrow[dr, phantom, "(2)"] & G_{12} \arrow[r, Rightarrow] \arrow[d, Rightarrow, bend right, "Q!"] \arrow[d, Rightarrow, bend left, "Q!"'] & \cdots \arrow[r, Rightarrow, "p_n"] \arrow[dr, phantom, "(n)"] & G_{1_n} = G_2 \arrow[d, Rightarrow, "Q!"'] \\
G_1^* \arrow[r, Rightarrow, "p_1^*"'] & G_{11}^* = G_{11}^+ \arrow[r, Rightarrow, "p_2^*"'] & G_{12}^* = G_{12}^+ \arrow[r, Rightarrow] & \cdots \arrow[r, Rightarrow, "p_n^*"'] & G_{1n}^* = G_2^*
\end{tikzcd}$$

Note that $G_{11} \overset{Q!}{\Longrightarrow} G_{11}^*$ and $G_{11} \overset{Q!}{\Longrightarrow} G_{11}^+$ are in general defined by different $Q$-sequences induced by $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ and $p_2 \overset{Q!}{\Longrightarrow} p_2^*$, respectively. But termination and local confluence of $Q$ by assumptions 1 and 2 implies unique normal forms and hence, $G_{11}^* = G_{11}^+$ (up to isomorphism), and similarly $G_{12}^* = G_{12}^+,..,G_{1_{n-1}}^* = G_{1_{n-1}}^+$.

Finally, $G_1^* \Longrightarrow G_2^*$ via $(p_1^*, .., p_n^*)$ is terminating by termination compatibility according to assumption 3. Hence, we have diagram $(A)$ for each $G_1 \in VL_1$, with $G_2 \in VL_2, G_1^* \in VL_1^*$ and $G_2^* \in VL_2^*$, where $VL_2^* = \{G_2^* | \exists G_2 \in VL_2 : G_2 \overset{Q!}{\Longrightarrow} G_2^*\}$, which implies terminating $(MR_2, Q) : VL_2 \to VL_2^*$ and $(MT^*, P^*) : VL_1^* \to VL_2^*$ with commutativity of diagram $(B)$:

$$\begin{tikzcd}
G_1 \arrow[r, Rightarrow, "P!"] \arrow[d, Rightarrow, "Q!"'] \arrow[dr, phantom, "(A)"] & G_2 \arrow[d, Rightarrow, "Q!"] \\
G_1^* \arrow[r, Rightarrow, "P^*!"'] & G_2^*
\end{tikzcd}$$

$$\begin{tikzcd}
VL_1 \arrow[r, "{(MT,P)}"] \arrow[d, "{(MR_1,Q)}"'] \arrow[dr, phantom, "(B)"] & VL_2 \arrow[d, "{(MR_2,Q)}"] \\
VL_1^* \arrow[r, "{(MT^*,P^*)}"'] & VL_2^*
\end{tikzcd}$$

*Remark 1.* If $(MT, P)$ and $(MT^*, P^*)$ are not functional, then commutativity of diagram $(B)$ means that for each $G_1 \overset{P!}{\Longrightarrow} G_2$ exists a corresponding $G_1^* \overset{P^*!}{\Longrightarrow} G_2^*$ such that diagram $(A)$ commutes.

*Example 7 (Refactoring of the SC2PN Model Transformation).*

In order to apply Theorem 1, we have to show the *required properties*:

1. The original model transformation $(MT, P) =$ *SC2PN* is terminating by [6]. The refactoring operation markState is terminating, because rules markInitial and markNormal delete one attribute each, and therefore each rule is only applicable once at a match to a State node. The refactoring of the model transformation rules $(MR, Q)$ is terminating, because at most one rule $q \in Q$ with $Q = \{markInitial, markNormal\}$ is applicable once.
2. The refactoring rules in $Q$ are locally confluent: rules markInitial and markNormal are parallel independent because their left-hand sides overlap in gluing point State only. Moreover, there is at most one match of markInitial resp. markNormal at the same State.
3. $Q$ is $(P, P^*)$-compatible as shown in Example 6.

According to the application of Theorem 1, we obtain the terminating model refactoring $(MR_2, Q)$, and the terminating model transformation $(MT^*, P^*)$ for each possible statechart which is transformed to a Petri net using $(MT, P)$, i.e. the rules in $P$, and which is refactored using the refactoring $(MR_1, Q)$, i.e. the rules in $Q$. As result we have the commutative diagram below, where $VL_1$ is the visual language of statecharts,
$VL_1^*$ is the statechart language, extended by the new node types Initial and Normal for the markState refactoring, $VL_2$ is the integrated language of statecharts and Petri nets (defined by the type graph in Fig. 1), and $VL_2^*$ is the integrated language of extended statecharts and Petri nets.

$$\begin{tikzcd}
VL_1 \arrow[r, "{(MT,P)}"] \arrow[d, "{(MR_1,Q)}"'] & VL_2 \arrow[d, "{(MR_2,Q)}"] \\
VL_1^* \arrow[r, "{(MT^*,P^*)}"'] & VL_2^*
\end{tikzcd}$$

# 5 Extensions of Main Results

## 5.1 General Model Refactoring Rules $Q$

We have assumed that $Q$-rules are nondeleting (bijective) on nodes. This was essential in Lemma 1 to construct the transformation $G_1 \overset{q}{\Longrightarrow} G_1'$.
In a direct proof of the main result, this can be avoided if we have parallel independence (with NACs) of all $P$- and $Q$-rules. By the Local Church-Rosser Theorem, this would lead to the diagram to the right, with $P^* = P$, where $Q$-rules are not applied to $P$.

$$\begin{tikzcd}
VL_1 \arrow[r, "{(MT,P)}"] \arrow[d, "{(MR_1,Q)}"'] & VL_2 \arrow[d, "{(MR_2,Q)}"] \\
VL_1^* \arrow[r, "{(MT^*,P)}"'] & VL_2^*
\end{tikzcd}$$

In our example, however, we do not have parallel independence of $P$- and $Q$-rules, but of $P^*$- and $Q$-rules, as required by $Q$-$(P, P^*$-$)$ compatibility. In fact, our refactoring rule $q$ is not parallel independent of the model transformation rule $p_1$ but parallel independent of the refactored model transformation rule $p_1^*$. This is also the case for all other refactored model transformation rules $p_i^*$ because $L_i^*$ does not contain the attribute "*IsInit = true*".

## 5.2 Model Refactoring Rules with NACs

We have assumed that $Q$-rules have no NACs. Now, we consider $Q$-rules which are still nondeleting on nodes, but with NACs. In Lemma 1, we assume to have $p_1 \xRightarrow{q,m} p_2$ with $m \models nac_q$ and have to show for $G_1 \overset{q,m_1 \circ m}{\Longrightarrow} G_1'$ and $G_2 \overset{q,g_1 \circ m_1 \circ m}{\Longrightarrow} G_2'$ that $m_1 \circ m \models nac_q$ implies $g_1 \circ m_1 \circ m \models nac_q$. This means, we have to require that $m \models nac_q$ implies $g_1 \circ m_1 \circ m \models nac_q$ because this also implies $m_1 \circ m \models nac_q$.

In Lemma 2, we need the following more general NAC-compatibility of $Q$: Whenever $G_{1_i} \overset{p_{1_i}}{\Longrightarrow} G_{2_i}$ is derivable from $G_1 \overset{p_1}{\Longrightarrow} G_2$ with $p_1 \in P$ and $p_{1_i} \overset{q_i,m_i}{\Longrightarrow} p_{1_{i+1}}$ satisfies $nac_{q_i}$, then also the extension of match $m_i : L_{q_i} \to L_{1_i}$ to $G_{1_i}$ and $G_{2_i}$ satisfies $nac_{q_i}$ for $i = 1, .., n$. Moreover, we need independence compatibility for rules $Q, P$ and $P^*$ with NACs. For the last step, we need local confluence of rules $Q$ with NACs. Both can be obtained from the corresponding Local Church-Rosser Theorem and the Local Confluence Theorem with NACs [18].

## 5.3 Extended Application of Refactoring Rules to Model Transformation Rules

In Def. 1, the application of a $Q$-rule $q$ to a $P$-rule $p_1 : L_1 \to R_1$ with $nac_1 : L_1 \to N_1$ was only possible if we had a match $m : L_q \to L_1$.
If this is not possible, we can also consider the case that we have a match $m : L_q \to R_1$ satisfying $nac_q$ and no $L_1$-deletion, i.e. we have the pushout-complement $E$ in (1) and $d : L_1 \to E$, such that (3) commutes (see the diagram to the right).

$$\begin{tikzcd}
 & L_q \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & I_q \arrow[l, "l"'] \arrow[r, "r"] \arrow[d] \arrow[dr, phantom, "(2)"] & R_q \arrow[d] \\
L_2 = L_1 \arrow[r, "p_1"] \arrow[rr, bend right, "d"', "(3)"] & R_1 & E \arrow[l, "r_1"'] \arrow[r, "r_2"] & R_2
\end{tikzcd}$$

In this case, the resulting rule $p_2$ is given by $p_2 : L_2 = L_1 \xrightarrow{d} E \xrightarrow{r_2} R_2$, where $r_2$ is defined by pushout (2). In this case we need more restrictive assumptions to obtain the main result.

## 6 Additional Refactoring Rules

In this section, we consider a few more refactorings for our example and validate the compatibility criteria discussed in Section 4.

### 6.1 Refactoring *State2SimpleState*

Fig. 8 shows refactoring rules for renaming State nodes into SimpleState nodes which may be applied after the refactoring in Section 2.2.

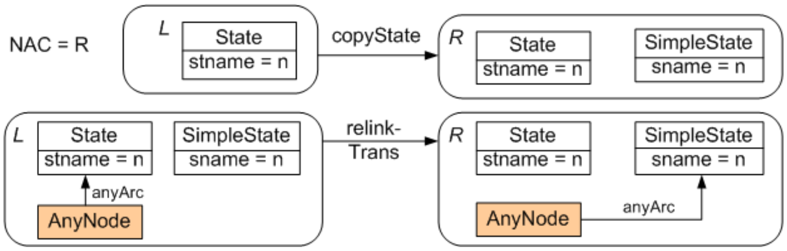


**Fig. 8.** Refactoring Rules for *State2SimpleState*

First, rule copyState creates new SimpleState nodes while the attribute value of stname is copied to sname. This rule is applied only once for each State with NAC=R. Secondly, rule relinkTrans removes any incoming arc from State and links it to the previously inserted SimpleState node. AnyNode should be treated as superclass of all nodes of the extended SC2PN type graph (i.e. replaced with Trans, Initial, Normal and RefState).

Fig. 9 shows two of the refactored model transformation rules after applying the *State2SimpleState* refactoring to the model transformation rules resulting of the *markState* refactoring.

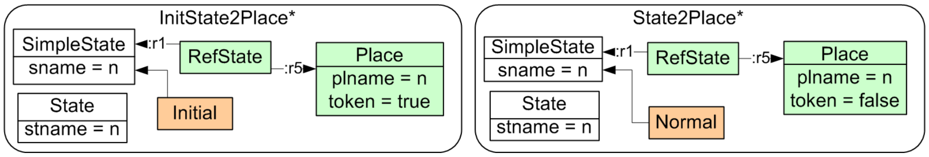


**Fig. 9.** Refactored Model Transformation Rules of *SC2PN* after *markState* and *State2SimpleState*

Finally, all isolated State nodes should be removed by restriction to the adapted type graph.

We first show that we have $Q$– $(P,\ P^*)$– compatibility (i.e. independence and termination compatibility) as defined in Def. 2:

- Independence compatibility: Given terminal $p_1^*$ wrt. $Q$ and $q \in Q$ with $G_1' \overset{q}{\Longleftarrow} G_1 \overset{p^*}{\Longrightarrow} G_2$, we must show that we have parallel independence. For $q = \mathit{copyState}$, we have the situation that there cannot be a graph $G$ where $q$ and any refactored model transformation rule $p_i^*$ (see e.g. Fig. 9) are both applicable: On the one hand, any $p_i^*$ is applicable only when a *State* and the corresponding *SimpleState* with the same value for their name attributes exist in the graph. On the other hand, the NAC of *copyState* forbids its application in this case.
  For $q = \mathit{relink}$, we have parallel independence for all rule $p_i^*$, as no rule deletes elements that are needed by the other rule. We do not have to consider NACs here.
- Termination compatibility: Given terminal $G$ wrt. $P$ and $G \overset{Q!}{\Longrightarrow} G^*$, then the *State2SimpleState* refactoring rules have been applied to all state nodes occuring in a rule in $P$, and to all state nodes in $G$. So there is no match from a rule $p^* \in P^*$ to $G^*$ where the NAC of $p^*$ would not prevent its application, and hence, $G^*$ is terminal wrt. $P^*$.

We now can show the required properties for applying Theorem 1:

1. We already know that the original model transformation SC2PN is terminating [6]. The refactoring operation *State2SimpleState* is terminating because of the NAC of rule *copyState*, and the arc replacement operation defined by rule *relink*, which can be applied exactly once for each existing arc pointing to a *State* node.
2. The refactoring rules in $Q$ are locally confluent since they are parallel independent for non-overlapping matches. For overlapping matches, rule relink can only be applied when rule copyState has been applied before.
3. $Q$ is $(P, P^*)$-compatible as shown above.

### 6.2 Refactoring *UnifyNames*

Fig. 10 shows refactoring rules for unifying the name attributes (stname and plname) from nodes State and Place to name. The old attribute name is deleted in the left-hand side of the rule while the new name is inserted on the right-hand side.

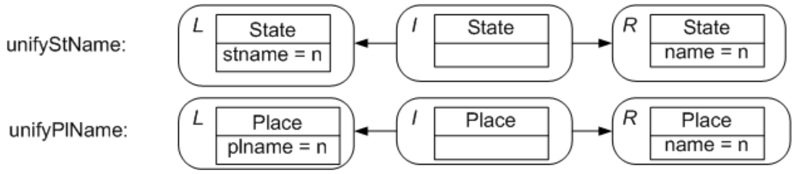


**Fig. 10.** Refactoring Rules for *UnifyNames*

Fig. 11 shows two of the refactored model transformation rules after applying the *UnifyNames* refactoring to the model transformation rules resulting of the *markState* refactoring.

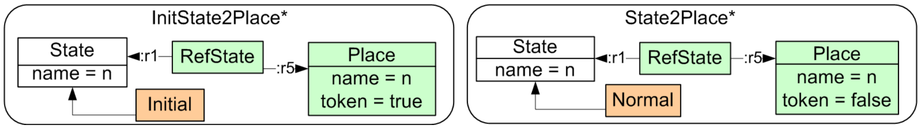


**Fig. 11.** Refactored Model Transformation Rules of *SC2PN* after *markState* and *UnifyNames*

Again, we first show that we have $Q$– $(P, P^*)$– compatibility (i.e. independence and termination compatibility) as defined in Def. 2:

- Independence compatibility: Given terminal $p_1^*$ wrt. $Q$ and $q \in Q$ with $G_1' \overset{q}{\Longleftarrow} G_1 \overset{p^*}{\Longrightarrow} G_2$, we must show that we have parallel independence. For $q =$ *unifySName*, we have the situation that there cannot be a graph $G$ where $q$ and any refactored model transformation rule $p_i^*$ (see e.g. Fig. 11) are both applicable: On the one hand, any $p_i^*$ is applicable only to a *State* with an attribute *name* assigned to $n$ and without an attribute *stname*. On the other hand, rule $unifySName$ is applicable only to a *State* with an attribute *stname* assigned to $n$. Analogously, $q =$ *unifySName* is parallel independent of all refactored model transformation rule $p_i^*$.
  For $q =$ *relink*, we have parallel independence for all rule $p_i^*$, as no rule deletes elements that are needed by the other rule. We do not have to consider NACs here.
- Termination compatibility: Given terminal $G$ wrt. $P$ and $G \overset{Q!}{\Longrightarrow} G^*$, then the *unifySName* refactoring rules have been applied to all state and place nodes occuring in a rule in $P$, and to all state and place nodes in $G$. So there is no match from a rule $p^* \in P^*$ to $G^*$ where the NAC of $p^*$ would not prevent its application, and hence, $G^*$ is terminal wrt. $P^*$.

We now can show the required properties for applying Theorem 1:

1. We already know that the original model transformation SC2PN is terminating [6]. The refactoring operation *UnifyName* is terminating because both rules are applicable as many times as there are *State* attributes of type *stname* and *Place* attributes of type *plname*.
2. The refactoring rules in $Q$ are locally confluent since they are parallel independent.
3. $Q$ is $(P, P^*)$-compatible as shown above.

## 7 Related Work

Refactoring of information systems is a common technique for software evolution through transformation [19, 3]. Automated transformation within domain specific languages including version support has been considered in [20, 21].

Refactoring by graph transformation rules plays an important role for software system refactoring by providing a graphical way for rule definition and an underlying algebraic framework for analyzing refactoring dependencies [12] and to assure behavior preservation in model refactoring using transformations with borrowed contexts [22]. Moreover suitable verification techniques are available, e.g. architectural refactoring by rule extraction [23].

From a technical point of view, in this paper we apply model refactoring rules $Q$ deleting (on edges) to non-deleting transformation rules $P$, which is in some sense dual to the $S2A$-construction of animation rules $P_A$ from simulation rules $P_S$ in [17], where non-deleting rules $Q$ are applied to deleting rules $P_S$. Both kinds of rule transformations are based on the construction in [14] but have been extended by NACs and by the possibility to transform generated or deleted rule objects, as well.

Within the Eclipse Modeling Framework [24] model refactoring has already been implemented using graph transformation concepts [25]. While software refactoring is a common technique, a general theory for refactoring of model transformations has still been missing.

## 8 Conclusion

In this paper, we consider a graph-transformation-based evolution of model transformations which adapts model transformation rules to refactored models. In the main result, we show that under suitable assumptions, the evolution leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to refactor a model transformation from statecharts to Petri nets.

As future research, we intend to consider refactoring operations at type graph level based on our approach on transformations of type graphs with inheritance [26]. Moreover, up to now, we have studied model transformations resulting in an integrated model which contains both source and target language elements. A restriction to the target model presently means that we get the same target model as before refactoring the source model and the model transformation rules. Additionally, we plan to handle target language refactorings analogously to refactorings of the source language.

# References

1. Beydeda, S., Book, M., Gruhn, V., eds.: Model-Driven Software Development. Springer-Verlag, Heidelberg (2005)
2. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
3. Mens, T., Tourwé, T.: A survey of software refactoring. Transactions on Software Engineering **30**(2) (2004) 126–139
4. Boger, M., Sturm, T., Fragemann, P.: Refactoring Browser for UML. In: Proc. 3rd Intl Conf. on eXtreme Programming and Flexible Processes in Software Engineering, Alghero, Sardinia. (2002) 77–81
5. Sunyé, G., Pollet, D., LeTraon, Y., Jézéquel, J.M.: Refactoring UML Models. In: Proc. UML 2001. Volume 2185 of LNCS., Heidelberg, Springer-Verlag (2001) 134–138
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theoretical Computer Science. Springer Verlag (2006)
7. Mens, T., Van Gorp, P., Varrò, D., Karsai, G.: Applying a Model Transformation Taxonomy to Graph Transformation Technology . In: Proc. International Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Elsevier Science (2005) 143–159
8. Königs, A.: Model Transformation with Triple Graph Grammars. In: Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica. (2005)
9. Ehrig, H., Ehrig, K.: Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In: Proc. International Workshop on Graph and Model Transformation (GraMoT'05). Volume 152 of ENTCS., Tallinn, Estonia, Elsevier Science (2005)
10. : AGG (2009) http://tfs.cs.tu-berlin.de/agg.
11. Mens, T., Taentzer, G., Müller, D.: Model-driven software refactoring. In Rech, J., Bunse, C., eds.: Model-Driven Software Development: Integrating Quality Assurance. Idea Group Inc. (2005) 170–203
12. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling **6**(3) (2007) 269–285
13. Grunske, L., Geiger, L., Zündorf, A., Van Eetvelde, N., Van Gorp, P., Varro, D.: Using Graph Transformation for Practical Model Driven Software Engineering. In Beydeda, S., Book, M., Gruhn, V., eds.: Model-driven Software Development. Springer (2005) 91–118
14. Parisi-Presicce, F.: Transformation of Graph Grammars. In: 5th Int. Workshop on Graph Grammars and their Application to Computer Science. Volume 1073 of LNCS., Springer (1996)
15. Ehrig, H., Ehrig, K., Ermel, C.: Evolution of model transformations by model refactoring. In: Proc. Workshop of Graph Transformation and Visual Modeling Techniques (GT-VMT'09). (2009)
16. Reisig, W.: Petri Nets: An Introduction. Volume 4 of EATCS Monographs on Theoretical Computer Science. Springer Verlag (1985)
17. Ehrig, H., Ermel, C.: Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation. In: Proc. International Conference on Graph Transformation (ICGT'08). Volume 5214 of LNCS., Heidelberg, Springer Verlag (2008) 194–210
18. Lambers, L., Ehrig, H., Prange, U., Orejas, F.: Embedding and Confluence of Graph Transformations with Negative Application Conditions. In Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G., eds.: Proc. International Conference on Graph Transformation (ICGT'08). Volume 5214 of LNCS., Heidelberg, Springer Verlag (2008) 162–177
19. Löwe, M., König, H., Peters, M., Schulz, C.: Refactoring Information Systems. In Favre, J.M., Heckel, R., Mens, T., eds.: Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Chance (SeTra 2006). Volume 3., Natal, Brazil, Electronic Communications of the EASST (2006)
20. Bell, P.: Automated Transformation of Statements within Evolving Domain Specific Languages. In Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.P., eds.: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling. Volume TR-38., Finland, Computer Science and Information System Reports, Technical Reports, University of Jyvskyl (2007)
21. de Geest, G., Savelkoul, A., Alikoski, A.: Building a framework to support Domain Specific Language evolution using Microsoft DSL Tools. In Sprinkle, J., Gray, J., Rossi, M., Tolvanen, J.P., eds.: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling. Volume TR-38., Finland, Computer Science and Information System Reports, Technical Reports, University of Jyvskyl (2007)
22. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior Preservation in Model Refactoring using DPO Transformations with Borrowed Contexts. In: Proc. International Conference on Graph Transformation (ICGT'08). Volume 5214 of LNCS., Heidelberg, Springer Verlag (2008)

23. Bisztray, D., Heckel, R., Ehrig, H.: Verification of Architectural Refactorings by Rule Extraction. In Fiadeiro, J., Inverardi, P., eds.: Proc. Fundamental Approaches to Software Engineering (FASE'08). Volume 4961 of LNCS., Springer Verlag (2008) 347–361
24. Eclipse Consortium: Eclipse Modeling Framework (EMF) – Version 2.4. (2008) http://www.eclipse.org/emf.
25. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In: Proc. 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06), Genova, Italy (2006)
26. Ehrig, H., Ermel, C., Hermann, F.: Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In Wirsing, M., Chechik, M., eds.: Proc. International Conference on Fundamental Aspects of Software Engineering (FASE'09). LNCS, Heidelberg, Springer Verlag (2009) To appear.