Behavior Preservation in Model Refactoring using DPO Transformations with Borrowed Contexts[*]

Guilherme Rangel[1], Leen Lambers[1], Barbara König[2], Hartmut Ehrig[1], and Paolo Baldan[3]

[1] Institut für Softwaretechnik und Theoretische Informatik,
Technische Universität Berlin, Germany
{rangel,leen,ehrig}@cs.tu-berlin.de

[2] Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany
barbara_koenig@uni-due.de

[3] Dipartimento di Matematica Pura e Applicata,
Università di Padova, Italy
baldan@math.unipd.it

Abstract. Behavior preservation, namely the fact that the behavior of a model is not altered by the transformations, is a crucial property in refactoring. The most common approaches to behavior preservation rely basically on checking given models and their refactored versions. In this paper we introduce a more general technique for checking behavior preservation of refactorings defined by graph transformation rules. We use double pushout (DPO) rewriting with borrowed contexts, and, exploiting the fact that observational equivalence is a congruence, we show how to check refactoring rules for behavior preservation. When rules are behavior-preserving, their application will never change behavior, i.e., every model and its refactored version will have the same behavior. However, often there are refactoring rules describing intermediate steps of the transformation, which are not behavior-preserving, although the full refactoring does preserve the behavior. For these cases we present a procedure to combine refactoring rules to behavior-preserving concurrent productions in order to ensure behavior preservation. An example of refactoring for finite automata is given to illustrate the theory.

1 Introduction

Model transformation [1] is concerned with the automatic generation of models from other models according to a transformation procedure which describes how a model in the source language can be "translated" into a model in the target language. Model refactoring is a special case of model transformation where the

[*] Research partially supported by DAAD (German Academic Exchange Service), DFG project Behavior-GT and MIUR project ART.

source and target are instances of the same metamodel. Software refactoring is a modern software development activity, aimed at improving system quality with internal modifications of source code which do not change the observable behavior. In object-oriented programming usually the observable behavior of an object is given by a list of public (visible) properties and methods, while its internal behavior is given by its internal (non-visible) properties and methods.

Graph transformation systems (GTS) are well-suited to model refactoring and, more generally, model transformation (see [2] for the correspondence between refactoring and GTS). Model refactorings based on GTS can be found in [3–6]. The left part of Fig. 1 describes schematically model refactoring via graph transformations. For a graph-based metamodel M, describing, e.g., deterministic finite automata or statecharts, the set $\mathit{Refactoring}^M$ of graph productions describes how to transform models which are instances of the metamodel M. A start graph G^M, which is an instance of the metamodel M, is transformed according to the productions in $\mathit{Refactoring}^M$ (using regular DPO transformations), thus producing a graph H^M which is the refactored version of G^M.

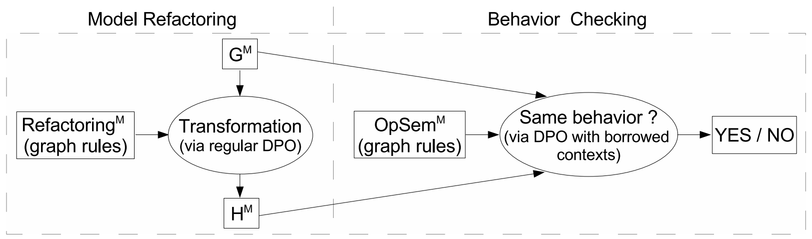
Model Refactoring
Behavior Checking
G^M
Refactoring^M (graph rules)
Transformation (via regular DPO)
H^M
OpSem^M (graph rules)
Same behavior ? (via DPO with borrowed contexts)
YES / NO

Fig. 1. Model refactoring via graph transformations and behavior preservation.

A crucial question that must be asked always is whether a given refactoring is behavior-preserving, which means that source and target models have the same observable behavior. In practice, proving behavior-preservation is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. On the other hand, formal approaches [7–10] have been also employed. A common issue is that behavior preservation is checked only for a certain number of models and their refactored versions. It is difficult though to foresee which refactoring steps are behavior-preserving for all possible instances of the metamodel. Additionally, these approaches are usually tailored to specific metamodels and the transfer to other metamodels would require reconsidering several details. A more general technique is proposed in [11] for analyzing the behavior of a graph production in terms of CSP processes and trace semantics which guarantees that the traces of a model are a subset of the traces of its refactored version.

In [3] we employed the general framework of borrowed contexts [12] to show that models are bisimilar to their refactored counterparts, which implies behavior preservation. The general idea is illustrated in the right-hand side of Fig. 1.

We define a set $OpSem^M$ of graph productions describing the operational semantics of the metamodel M and use the borrowed context technique to check whether the models G^M and H^M have the same behavior w.r.t. $OpSem^M$. In [3] we also tailored Hirschkoff's up-to bisimulation checking algorithm [13] to the borrowed context setting and thus equivalence checking can in principle be carried out automatically. The main advantage of this approach is that for every metamodel whose operational semantics can be specified in terms of finite graph transformation productions, the bisimulation checking algorithm can be used to show bisimilarity between models which are instances of this metamodel. However, this technique is also limited to showing behavior preservation only for a fixed number of instances of a metamodel.

In this paper we go a step further and employ the borrowed context framework in order to check refactoring productions for behavior preservation according to the operational semantics of the metamodel. We call a rule behavior-preserving when its left- and right-hand sides are bisimilar. Thanks to the fact that bisimilarity is a congruence, whenever all refactoring productions preserve behavior, so does every transformation via these rules. In this case, all model instances of the metamodel and their refactored versions exhibit the same behavior. However, refactorings very often involve non-behavior-preserving rules describing intermediate steps of the whole transformation. Given a transformation $G \overset{p_1}{\Rightarrow} H$ via a non-behavior-preserving rule p_1, the basic idea is then to check for the existence of a larger transformation $G \Rightarrow^* H'$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rule applications such that the concurrent production [14, 15] induced by seq is behavior-preserving. Since the concurrent production p_c performs exactly the same transformation $G \overset{p_c}{\Rightarrow} H'$ we can infer that G and H' have the same behavior.

This paper is structured as follows. Section 2 briefly reviews how the DPO approach with borrowed contexts can be used to define the operational semantics of a metamodel. Section 3 defines the model refactorings we deal with. An example in the setting of finite automata is given in Section 4. In Section 5 we define a technique to check refactoring rules for behavior preservation and an extension to handle non-behavior-preserving rules in model refactoring. Finally, these techniques are applied to the automata example. The proofs of the results in this paper, which are omitted here because of space limitations, can be found in [16].

2 Operational Semantics via Borrowed Contexts

In this section we recall the DPO approach with borrowed contexts [12, 17] and show how it can be used to define the operational semantics of a metamodel M. In this paper we consider the category of labeled graphs, but the results would also hold for the category of typed graphs or, more generally, for adhesive categories. In standard DPO [18], productions rewrite graphs with no interaction with any other entity than the graph itself. In the DPO approach with borrowed contexts [17] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

Definition 1 (Graphs with Interfaces and Graph Contexts). *A* graph G with interface J *is a morphism* $J \to G$ *and a context consists of two morphisms* $J \to E \leftarrow \overline{J}$*. The* embedding *of a graph with interface* $J \to G$ *into a context* $J \to E \leftarrow \overline{J}$ *is a graph with interface* $\overline{J} \to \overline{G}$ *which is obtained by constructing* $\overline{G}$ *as the pushout of* $J \to G$ *and* $J \to E$*.*

$$\begin{tikzcd}
J \arrow[r] \arrow[d] \arrow[rd, phantom, "PO"] & E \arrow[d] & \overline{J} \arrow[l] \arrow[ld, dotted] \\
G \arrow[r] & \overline{G} &
\end{tikzcd}$$

Observe that the embedding is defined up to isomorphism since the pushout object is unique up to isomorphism.

Definition 2 (Metamodel M and Model). *A* metamodel M *specifies a set of graphs with interface of the form* $J \to G$ *(as in Definition 1). An element of this set is called an instance of the metamodel* M*, or simply* model*.*

For example, the metamodel *DFA*, introduced in Section 4, describes deterministic finite automata. A model is an automaton $J \to G$, where G is the automaton and J specifies which parts of G may interact with the environment.

Definition 3 (Set of Operational Semantics Rules). *Given a metamodel* M *as in Definition 2, its operational semantics is defined by a set* $OpSem^M$ *of graph productions* $L \xleftarrow{l} I \xrightarrow{r} R$*, where* l, r *are injective morphisms.*

Definition 4 (Rewriting with Borrowed Contexts). *Let* $OpSem^M$ *be as in Definition 3. Given a model* $J \to G$ *and a production* $p\colon L \leftarrow I \to R$ *(*$p \in OpSem^M$*), we say that* $J \to G$ *reduces to* $K \to H$ *with transition label* $J \to F \leftarrow K$ *if there are graphs* D, G^+, C *and additional morphisms such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with injective morphisms. In this case a* rewriting step with borrowed context *(BC step) is called feasible:* $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$*.*

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[rd, phantom, "PO"] & L \arrow[d] \arrow[rd, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[rd, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] \arrow[ru, phantom, "PO"] & F \arrow[u] \arrow[ru, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ru, dotted] &
\end{tikzcd}$$

In the diagram above the upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 4, namely C and F, may not exist. In this case, the rewriting step is not feasible. Furthermore, observe that for a given partial match $G \leftarrow D \rightarrow L$ the graphs G^+ and C are uniquely determined.

A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

Definition 5 (Bisimulation and Bisimilarity). *Let $OpSem^M$ be as in Definition 3 and let $\mathcal{R}$ be a symmetric relation containing pairs of models $(J \rightarrow G, J \rightarrow G')$. The relation $\mathcal{R}$ is called a* bisimulation *if, whenever we have $(J \rightarrow G)\,\mathcal{R}\,(J \rightarrow G')$ and a transition $(J \rightarrow G) \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H)$, then there exists a model $K \rightarrow H'$ and a transition $(J \rightarrow G') \xrightarrow{J \rightarrow F \leftarrow K} (K \rightarrow H')$ such that $(K \rightarrow H)\,\mathcal{R}\,(K \rightarrow H')$.*

We write $(J \rightarrow G) \sim_{OpSem^M} (J \rightarrow G')$ (or $(J \rightarrow G) \sim (J \rightarrow G')$ if the operational semantics is obvious from the context) whenever there exists a bisimulation $\mathcal{R}$ that relates the two instances of the metamodel M. The relation $\sim_{OpSem^M}$ is called bisimilarity.

When defining the operational semantics using the borrowed context framework, it should be kept in mind that rewriting is based on interactions with the environment, i.e., the environment should provide some information via F to the graph G in order to trigger the rewriting step. For instance, in our finite automata example in Section 4 the environment provides a letter to trigger the corresponding transition of the automaton.

An advantage of the borrowed context technique is that the derived bisimilarity is automatically a congruence, which means that whenever a graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace one part of the model by another bisimilar one, without altering its observable behavior.

Theorem 1 (Bisimilarity is a Congruence [12]). *Bisimilarity $\sim$ is a congruence, i.e., it is preserved by embedding into contexts as given in Definition 1.*

In [17] a technique is defined to speed up bisimulation checking, which allows us to take into account only certain labels. A label is considered superfluous and called *independent* if we can add two morphisms $D \rightarrow J$ and $D \rightarrow I$ to the diagram in Definition 4 such that $D \rightarrow I \rightarrow L = D \rightarrow L$ and $D \rightarrow J \rightarrow G = D \rightarrow G$. That is, intuitively, the graph G to be rewritten and the left-hand side L overlap only in their interfaces. Transitions with independent labels can be ignored in the bisimulation game, since a matching transition always exists.

3 Refactoring Transformations

Here we define refactoring transformations using DPO rules with negative application conditions (NAC).

Definition 6 (NAC, Rule with NAC and Transformation). *A* negative application condition *$NAC(n)$ on L is an injective morphism $n\colon L \to NAC$. An injective match $m\colon L \to G$ satisfies $NAC(n)$ on L if and only if there is no injective morphism $q\colon NAC \to G$ with $q \circ n = m$.*

$$\begin{tikzcd} NAC \arrow[dr, "q"'] & L \arrow[l, "n"'] \arrow[d, "m"] \arrow[dl, phantom, "="] \\ & G \end{tikzcd} \qquad \begin{tikzcd} NAC & L \arrow[l] \arrow[d, "m"'] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\ & G_0 & C_0 \arrow[l] \arrow[r] & G_1 \end{tikzcd}$$

A negative application condition $NAC(n)$ is called satisfiable *if n is not an isomorphism.*

A rule *$L \overset{l}{\leftarrow} I \overset{r}{\rightarrow} R$ (l, r injective)* with NACs *is equipped with a finite set of negative application conditions $NAC_L = \{NAC(n_i) \mid i \in I\}$. A* direct transformation *$G_0 \overset{p,m}{\Longrightarrow} G_1$ via a rule p with NACs and an injective match $m\colon L \to G_0$ consists of the double pushout diagram above, where m satisfies all NACs of p.*

Note that if $NAC(n)$ is satisfiable then the identity match $id : L \to L$ satisfies $NAC(n)$. We will assume that for any rule with NACs, the corresponding negative application conditions are all satisfiable, so that the rule is applicable to at least one match (the identity match on its left-hand side).

Definition 7 (Layered Refactoring System and Refactoring Rule). *Let metamodel M be as in Definition 2. A* refactoring rule *is a graph rule as in Definition 6. A* layered refactoring system *$Refactoring^M$ for the metamodel M consists of k sets $Refactoring_i^M$ ($0 \leq i \leq k-1$) of refactoring rules. Each set $Refactoring_i^M$ defines a transformation layer.*

Definition 8 (Refactoring Transformation). *Let $Refactoring^M$ be as in Definition 7. A* refactoring transformation *$t\colon (J \to G_0) \Rightarrow^* (J \to G_n)$ is a sequence $(J \to G_0) \overset{p_1}{\Rightarrow} (J \to G_1) \overset{p_2}{\Rightarrow} \cdots \overset{p_n}{\Rightarrow} (J \to G_n)$ of direct transformations (as in Definition 6) such that $p_i \in Refactoring^M$ and t preserves the interface J, i.e., for each i ($0 \leq i < n$) there exists an injective morphism $J \to C_i$ with $J \to G_i = J \to C_i \to G_i$ (see diagram below). Moreover, in t each layer applies its rules as long as possible before the rules of the next layer come into play.*

$$\begin{tikzcd} NAC & L \arrow[l] \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\ & G_i & C_i \arrow[l] \arrow[r] & G_{i+1} \\ & J \arrow[u] \arrow[ur, bend right, "="] \arrow[urr, dotted, bend right] & & \end{tikzcd}$$

Note that refactoring transformations operate only on the internal structure of G_i while keeping the original interface J.

4 Example: Deleting Unreachable States in *DFA*

In this section we present an example of refactoring in the setting of deterministic finite automata (DFA). The metamodel *DFA* describes finite automata represented as graphs with interface as $\mathsf{J} \rightarrow \mathsf{DFA1}$ and $\mathsf{J} \rightarrow \mathsf{DFA2}$ in Fig. 2, where unlabeled nodes are states and directed labeled edges are transitions. An FS-loop marks a state as final. A W-node has an edge pointing to the current state and this edge points initially to the start state. The W-node is the interface, i.e., the only connection to the environment.

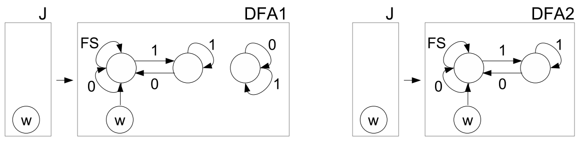

Fig. 2. Examples of DFA as graphs with interface.

The operational semantics for DFA is given by a set $\mathsf{OpSem}^{\mathsf{DFA}}$ of rules containing Jump(a), Loop(a) and Accept depicted in Fig. 3. The rules Jump(a), Loop(a) must be defined for each symbol $a \in \Lambda$, where Λ is a fixed alphabet. According to $\mathsf{OpSem}^{\mathsf{DFA}}$ a DFA may change its state. The W-node receives a symbol (e.g. 'b') from the environment in form of a b-labeled edge connecting W-nodes, e.g., the string 'bc' is $\textcircled{\mathsf{w}} \xleftarrow{\mathsf{b}} \textcircled{\mathsf{w}} \xleftarrow{\mathsf{c}} \textcircled{\mathsf{w}}$. An acpt-edge between W-nodes marks the end of a string. When such an edge is consumed by a DFA, the string previously processed is accepted.

A layered refactoring system for the deletion of unreachable states of an automaton is given in Fig. 3 on the right. To the left of each rule we depict the NAC (if it exists). The rules are spread over three layers. Rule1 marks the initial state as reachable with an R-loop. Rule2(a) identifies all other states that can be reached from the start state via a-transitions. Layer 1 deletes the loops and the edges of the unreachable states and finally the unreachable states. Layer 2 removes the R-loops.

Applying the refactoring rules above to the automaton $\mathsf{J} \rightarrow \mathsf{DFA1}$ we obtain $\mathsf{J} \rightarrow \mathsf{DFA2}$, where the rightmost state was deleted. By using the bisimulation checking algorithm of [3] we conclude that $\mathsf{J} \rightarrow \mathsf{DFA1}$ and $\mathsf{J} \rightarrow \mathsf{DFA2}$ are bisimilar w.r.t. $\mathsf{OpSem}^{\mathsf{DFA}}$. In our setting bisimilarity via the borrowed context technique corresponds to bisimilarity on automata seen as transition systems, which in turn implies language equivalence.

5 Behavior Preservation in Model Refactoring

Here we introduce a notion of behavior preservation for refactoring rules and, building on this, we provide some techniques for ensuring behavior preservation in model refactoring.

5.1 Refactoring via Behavior-Preserving Rules

For a metamodel M as in Definition 2 we define behavior preservation as follows.

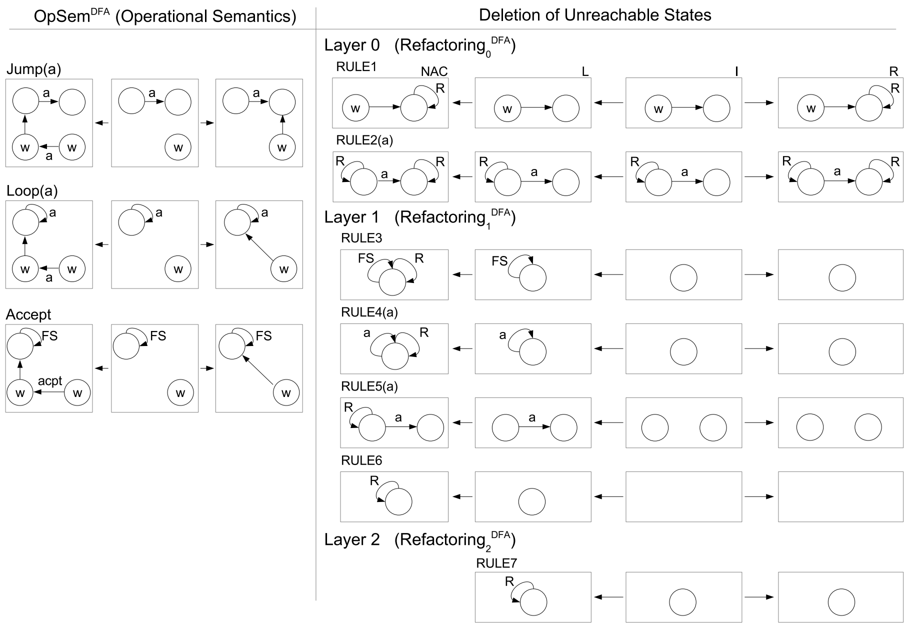
OpSem^DFA (Operational Semantics)
Deletion of Unreachable States
Jump(a)
Loop(a)
Accept
acpt
FS
Layer 0 (Refactoring_0^DFA)
Layer 1 (Refactoring_1^DFA)
Layer 2 (Refactoring_2^DFA)
RULE1
NAC
L
I
R
RULE2(a)
RULE3
RULE4(a)
RULE5(a)
RULE6
RULE7

Fig. 3. Operational semantics and a refactoring for DFA.

Definition 9 (Behavior-Preserving Transformation). *Let $OpSem^M$ be as in Definition 3. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 8) is called behavior-preserving when $(J \to G) \sim_{OpSem^M} (J \to H)$.*

In order to check t for behavior preservation we can use Definition 4 to derive transition labels from $J \to G$ and $J \to H$ w.r.t. the rules in $OpSem^M$.

Observe that behavior preservation in the sense of Definition 9 is limited to checking specific models. This process is fairly inefficient and can never be exhaustive as behavior-preservation must be checked for each specific transformation. A more efficient strategy consists in focussing on the behavior-preservation property at the level of refactoring rules. The general idea is to check for every $p \in Refactoring^M$ whether its left and right-hand sides, seen as graphs with interfaces, are bisimilar, i.e., $(I \to L) \sim (I \to R)$ w.r.t. $OpSem^M$. Whenever this happens, since bisimilarity is a congruence, any transformation $(J \to G) \overset{p}{\Rightarrow} (J \to H)$ via p will preserve the behavior, i.e., $J \to G$ and $J \to H$ have the same behavior.

Definition 10 (Behavior-Preserving Refactoring Rule). *Let $OpSem^M$ be as in Definition 3. A refactoring production $p\colon L \leftarrow I \to R$ with NAC_L is* behavior-preserving *whenever $(I \to L) \sim (I \to R)$ w.r.t. $OpSem^M$.*

Now we can show a simple but important result that says that a rule is behavior-preserving if and only if every refactoring transformation generated by this rule is behavior-preserving.

Proposition 1. *Let $OpSem^M$ be as in Definition 3. Then it holds: $p\colon L \leftarrow I \rightarrow R$ (with NAC_L) is behavior-preserving w.r.t. $OpSem^M$ if and only if any refactoring transformation $(J \rightarrow G) \overset{p}{\Rightarrow} (J \rightarrow H)$ (as in Definition 8) is behavior-preserving, i.e., $(J \rightarrow G) \sim_{OpSem^M} (J \rightarrow H)$.*

Remark 1. The fact that the previous proposition also holds for rules with NACs, even though Definition 10 does not take NACs into account for behavior-preservation purposes, does of course not imply that negative application conditions for refactoring rules are unnecessary in general. They are needed in order to constrain the applicability of rules, especially of those rules that are not behavior-preserving, or rather, are only behavior-preserving when applied in certain contexts. As a direction of future work, we plan to study congruence results for restricted classes of contexts. This will help to better handle refactoring rules with NACs.

Theorem 2 (Refactoring via Behavior-Preserving Rules). *Let $OpSem^M$ and $Refactoring^M$ be as in Definitions 3 and 7. If each rule in $Refactoring^M$ is behavior-preserving w.r.t. $OpSem^M$ then any refactoring transformation $(J \rightarrow G_0) \Rightarrow^* (J \rightarrow G_n)$ via these rules is behavior-preserving.*

Example 1. We check the rules in $\mathsf{Refactoring}_\mathsf{i}^{DFA}$ ($i = 0, 1, 2$) from Section 4 for behavior preservation. We begin with $\mathsf{Refactoring}_\mathsf{0}^{DFA}$ (Layer 0). For RULE1: $\mathsf{NAC} \leftarrow \mathsf{L} \leftarrow \mathsf{I} \rightarrow \mathsf{R}$ we derive the transition labels from $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ w.r.t. $\mathsf{OpSem}^\mathsf{DFA}$. On the left-hand side of Fig. 4 we schematically depict the first steps in their respective labeled transition systems (LTS), where each partner has three choices. Independent labels exist in both LTSs but are not illustrated below.

The derivation of label L_1 for $\mathsf{I} \rightarrow \mathsf{R}$ is shown on the right. Since $\mathsf{I} \rightarrow \mathsf{L}$ and $\mathsf{I} \rightarrow \mathsf{R}$ (and their successors) can properly mimic each other via a bisimulation we can conclude that $(\mathsf{I} \rightarrow \mathsf{L}) \sim_{\mathsf{OpSem}^\mathsf{DFA}} (\mathsf{I} \rightarrow \mathsf{R})$. The intuitive reason for this is that the R-loop, which is added by this rule, does not have any meaning in the operational semantics and is hence "ignored" by $\mathsf{OpSem}^\mathsf{DFA}$.

Analogously, RULE2(a) and the rule in Layer 2 are behavior-preserving as well. Hence, we can infer that every transformation via the rules of Layer 0 and Layer 2 preserves the behavior. On the other hand, all rules in Layer 1, except for RULE6, are not behavior-preserving. Note that RULE6 is only behavior-preserving because of the dangling condition. Thus, when a transformation is carried out via non-behavior-preserving rules of Layer 1 we cannot be sure whether the behavior has been preserved.

5.2 Handling Non-Behavior-Preserving Rules

For refactoring transformations based on non-behavior-preserving rules the technique of Section 5.1 does not allow to establish if the behavior is preserved.

Very often there are refactoring rules representing intermediate transformations that indeed are not behavior-preserving. Still, when considered together with neighboring rules, they could induce a concurrent production [14, 15] p_c,

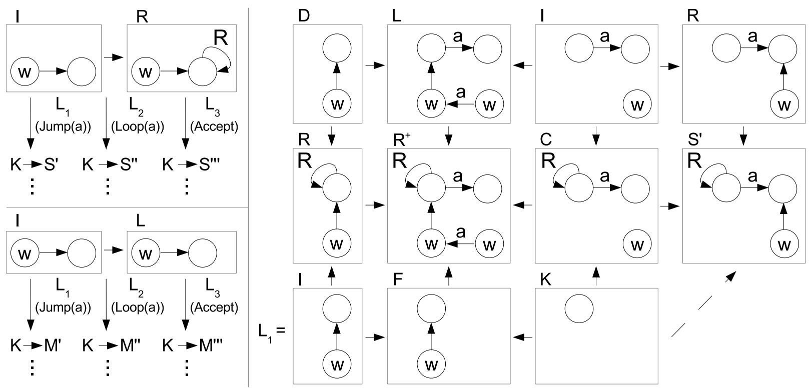

Fig. 4. Labeled transition systems for rule1 and a label derivation.

corresponding to a larger transformation, which preserves the behavior. For a transformation $t\colon (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ the concurrent production $p_c\colon L_c \leftarrow I_c \to R_c$ with concurrent NAC_{L_c} induced by t performs exactly the same transformation $(J \to G) \overset{p_c}{\Rightarrow} (J \to H')$ in one step. Moreover, p_c can only be applied to $(J \to G)$ if the concurrent NAC_{L_c} is satisfied. This is the case if and only if every NAC of the rules in t is satisfied. The basic idea is now to check for a transformation $(J \to G) \overset{p_1}{\Rightarrow} (J \to H)$ based on a non-behavior-preserving rule p_1 whether there exists such a larger transformation $t : (J \to G) \Rightarrow^* (J \to H')$ via a sequence $seq = p_1, p_2, \ldots, p_i$ of rules such that the concurrent production induced by t is behavior preserving. Then we can infer that $J \to G$ and $J \to H'$ have the same behavior.

This is made formal by the notion of safe transformation and the theorem below.

Definition 11 (Safe Transformation). *Let $OpSem^M$ be as in Definition 3. A refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ (as in Definition 8) is called* safe *if it induces a behavior-preserving concurrent production w.r.t. $OpSem^M$.*

Theorem 3 (Safe Transformations preserve Behavior). *Let $OpSem^M$ and $Refactoring^M$ be as in Definitions 3 and 7, and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. If t is safe, then t is behavior-preserving, i.e., $(J \to G) \sim (J \to H)$.*

In order to prove that a refactoring transformation $t\colon (J \to G) \Rightarrow^* (J \to H)$ is safe (and thus behavior-preserving), we can look for a split $t^{sp}\colon G \Rightarrow^* H_1 \Rightarrow^* \cdots \Rightarrow^* H_n \Rightarrow^* H$ (interfaces are omitted) of t where each step $(\Rightarrow^*)$ induces a behavior-preserving concurrent production (see Definition 12). In fact, as shown below, if and only if such split exists we can guarantee that t preserves behavior (Theorem 4).

Definition 12 (Safe Transformation Split). *Let $OpSem^M$ be as in Definition 3 and let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation (as in*

Definition 8). A split *of t is obtained by cutting t into a sequence of non-empty subtransformations $t^{sp}\colon (J \to G) \Rightarrow^* (J \to H_1) \Rightarrow^* \cdots \Rightarrow^* (J \to H_n) \Rightarrow^* (J \to H)$. A transformation split t^{sp} is* safe *if each step ($\Rightarrow^*$) is safe.*

In Section 5.3 we present a simple search strategy for safe splits. More elaborate ones are part of future work.

Theorem 4. *Let $t\colon (J \to G) \Rightarrow^* (J \to H)$ be a refactoring transformation. Then t is safe if and only if it admits a safe split.*

Observe that, instead, the following does not hold in general: if $t\colon (J \to G) \Rightarrow^* (J \to H)$ and $(J \to G) \sim_{OpSem^M} (J \to H)$ then t is safe. Consider for instance RULE5(a) in Fig. 3. As remarked, it is in general not behavior-preserving, but when, by coincidence, it removes a transition that is unreachable from the start state, the original automaton and its refactored version are behaviorally equivalent.

5.3 Ensuring Behavior Preservation

In this section we describe how the theory presented in this paper can be applied. Note that our results would allow us to automatically prove behavior preservation only in special cases, while, in general, such mechanized proofs will be very difficult. Hence here we will suggest a "mixed strategy", which combines elements of automatic verification and the search for behavior-preserving rules, in order to properly guide refactorings.

More specifically, a given model $J \to G$ can be refactored by applying the rules in $\mathit{Refactoring}^M$ in an automatic way, where the machine chooses non-deterministically the rules to be applied, or in a user-driven way, where for each transformation the machine provides the user with a list of all applicable rules together with their respective matches and ultimately the user picks one of them. The main goal is then to tell the user whether the refactoring is behavior-preserving.

The straightforward strategy to accomplish the goal above is to transform $J \to G$ applying only behavior-preserving rules. This obviously guarantees that the refactoring preserves behavior. However if a non-behavior-preserving rule p is applied we can no longer guarantee behavior preservation. Still, by proceeding with the refactoring, namely by performing further transformations, we can do the following: for each new transformation added to the refactoring we compute the induced concurrent production for the transformation which involves the first non-behavior-preserving rule p and the subsequent ones. If this concurrent production is behavior-preserving we can again guarantee behavior preservation for the refactoring since the refactoring admits a safe split (see Theorem 4).

The strategy above is not complete since behaviour preservation could be ensured by the existence of complex safe splits which the illustrated procedure is not able to find. We already have preliminary ideas for more sophisticated search strategies, but they are part of future work. Note however, that this strategy can reduce the proof obligations, since we do not have to show behavior preservation between the start and end graph of the refactoring sequence (which may be huge), but we only have to investigate local updates of the model.

Example 2. Consider the automaton $\mathsf{J} \to \mathsf{DFA1}$ of Section 4. By applying the behavior-preserving rules of $\mathsf{Refactoring}_0^{DFA}$ ($\mathsf{Layer\ 0}$) we obtain $\mathsf{J} \to \mathsf{DFA}_1^0$ depicted in Fig. 5 (the interface J is omitted). Since $\mathsf{Refactoring}_0^{DFA}$ contains only behavior-preserving rules by Theorem 2 it holds that $(\mathsf{J} \to \mathsf{DFA1}) \Rightarrow^* (\mathsf{J} \to \mathsf{DFA}_1^0)$ preserves the behavior. No more rules in $\mathsf{Refactoring}_0^{DFA}$ can be applied, i.e., the computation of $\mathsf{Layer\ 0}$ terminates.

Now the rules of $\mathsf{Refactoring}_1^{DFA}$ ($\mathsf{Layer\ 1}$) come into play. Recall that all rules in $\mathsf{Refactoring}_1^{DFA}$ are non-behavior-preserving, except for $\mathsf{RULE6}$. This set contains $\mathsf{RULE4(0)}$ and $\mathsf{RULE4(1)}$ which are appropriate instantiations of $\mathsf{RULE4(a)}$. After the transformation $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1)$ we can no longer guarantee behavior-preservation since $\mathsf{RULE4(0)}$ has been applied. From now on we follow the strategy previously described to look for a behavior-preserving concurrent production. We perform the step $(\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$, build a concurrent production p_c induced by $(\mathsf{J} \to \mathsf{DFA}_1^0) \overset{\mathsf{RULE4(0)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^1) \overset{\mathsf{RULE4(1)}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^2)$ and, by checking p_c for behavior-preservation, we find out that it is not behavior-preserving. We then continue with $(\mathsf{J} \to \mathsf{DFA}_1^2) \overset{\mathsf{RULE6}}{\Longrightarrow} (\mathsf{J} \to \mathsf{DFA}_1^3)$, build p'_c (Fig. 6), induced by the transformation beginning at $\mathsf{J} \to \mathsf{DFA}_1^0$ and check it for behavior-preservation. Now p'_c is behavior-preserving and so we can once again guarantee behavior preservation (Theorem 3).

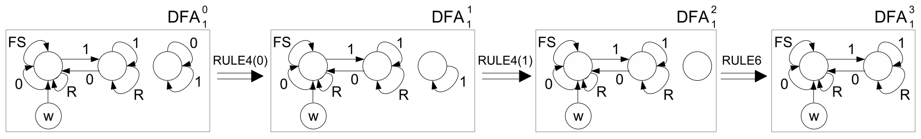

Fig. 5. Refactoring transformation.

Finally, no more rules of $\mathsf{Refactoring}_1^{DFA}$ are applicable to $\mathsf{J} \to \mathsf{DFA}_1^3$. The behavior-preserving rule in $\mathsf{Refactoring}_2^{DFA}$ ($\mathsf{Layer\ 2}$) comes into play and performs a transformation $(\mathsf{J} \to \mathsf{DFA}_1^3) \overset{\mathsf{RULE6}}{\Longrightarrow}_2 (\mathsf{J} \to \mathsf{DFA2})$, where the final automaton is depicted in Section 4 ($\mathsf{DFA2}$). Concluding, since we have found a safe split for the transformation via non-behavior-preserving rules we can infer that $\mathsf{J} \to \mathsf{DFA1}$ and $\mathsf{J} \to \mathsf{DFA2}$ have the same behavior.

Intuitively, the concurrent production is behavior-preserving, since it deletes an entire connected component that is not linked to the rest of the automaton. Note that due to the size of the components involved it can be much simpler to check such transformation units rather than the entire refactoring sequence.

In addition, it would be useful if the procedure above could store the induced concurrent productions which are behavior-preserving into $\mathit{Refactoring}^M$

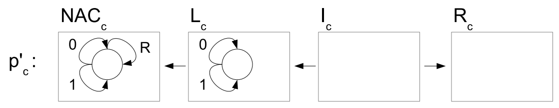

Fig. 6. Induced concurrent production p_c'.

for later use. By doing so the user knows which combination of rules leads to behavior-preserving concurrent productions. Similarly, the user could also want to know which combination of rules leads to non-behavior-preserving concurrent productions. Of course, in the latter case the concurrent productions are just stored but do not engage in any refactoring transformation. It is important to observe that we store into $\mathit{Refactoring}^M$ only concurrent productions which are built with rules within the same layer (as in Example 2). For more complex refactorings, such as the flattening of hierarchical statecharts (see [19]), a behavior-preserving concurrent production p_c exists only when it is built from a transformation involving several layers. In this latter case, p_c is built and checked for behavior preservation but not stored for later use.

For the cases where a layer $\mathit{Refactoring}_i^M$ of $\mathit{Refactoring}^M$ is terminating and confluent it is then important to guarantee that adding concurrent productions to the refactoring layer does not affect these properties.

Theorem 5. *Let* $\mathit{Refactoring}_i^M$ *be as in Definition 7 and* $R_i^{p_c}$ *be a set containing concurrent productions* p_c *built from* $p, q \in \mathit{Refactoring}_i^M \cup R_i^{p_c}$*. Then whenever* $\mathit{Refactoring}_i^M$ *is confluent and terminating it holds that* $\mathit{Refactoring}_i^M \cup R_i^{p_c}$ *is also terminating and confluent.*

For the case where layer $\mathit{Refactoring}_i^M$ is terminating and confluent another interesting and useful fact holds: assume that we fix a start graph G_0 and we can show that *some* (terminating) transformation, beginning with G_0 allows a behavior-preserving split. Then clearly all transformations starting from G_0 are behavior-preserving since they result in the same final graph H.

6 Conclusions and Future Work

We have shown how the borrowed context technique can be used to reason about behavior-preservation of refactoring rules and refactoring transformations. In this way we shift the perspective from checking specific models to the investigation of the properties of the refactoring rules.

The formal techniques in related work [7–10] address behavior preservation in model refactoring, but are in general tailored to a specific metamodel and limited to checking the behavior of a fixed number of models. Therefore, the transfer to different metamodels is, in general, quite difficult.

Hence, with this paper we propose to use the borrowed context technique in order to consider any metamodel whose operational semantics can be given by graph productions. Furthermore, the bisimulation checking algorithm [3] for borrowed contexts provides the means for automatically checking models for behavior preservation. This can be done not only for a specific model and its refactored version, but also for the left-hand and right-hand sides of refactoring rules. Once we have shown that a given rule is behavior-preserving, i.e., its left- and right-hand sides are equivalent, we know that its application will always preserve the behavior, due to the congruence result. When rules are not behavior-preserving, they still can be combined into behavior-preserving concurrent productions. We believe that such a method will help the user to gain a better understanding of the refactoring rules since he or she can be told exactly which rules may modify the behavior during a transformation. An advantage of our technique over the one in [11] is that we work directly with graph transformations and do not need any auxiliary encoding. Furthermore, with our technique we can guarantee that a model and its refactored version have exactly the same observable behavior, while in [11] the refactored model "contains" the original model but may add extra behavior.

This work opens up several possible directions for future investigations. First, in some refactorings when non-behavior-preserving rules are applied, the search strategies for safe splits can become very complex. Here we defined only a simple search strategy, but it should be possible to come up with more elaborate ones.

Second, although we are working with refactoring rules with negative application conditions, these NACs do not play a prominent role in our automatic verification techniques, but of course they are a key to limiting the number of concurrent productions which can be built. In [20] the borrowed context framework and the congruence result has been extended to handle rules with NACs. However, this applies only to negative application conditions in the operational semantics. It is, nevertheless, also important to have similar results for refactoring rules with NACs, which would lead to a "restricted" congruence result, where bisimilarity would only be preserved by certain contexts (see also the discussion in Remark 1). Since model refactorings often use graphs with attributes it is useful to check whether the congruence results in [12, 20] also hold for adhesive HLR categories (the category of attributed graphs is an instance thereof).

Acknowledgements: We would like to thank Gabi Taentzer and Reiko Heckel for helpful discussions on this topic.

References

1. Mens, T., Gorp, P.V.: A taxonomy of model transformation. ENTCS **152** (2006) 125–142
2. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) pp. 126–139
3. Rangel, G., König, B., Ehrig, H.: Bisimulation verification for the DPO approach with borrowed contexts. In: Proc. of GT-VMT '07. Volume 6 of Electronic Communications of the EASST. (2007)
4. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF model refactoring based on graph transformation concepts. In: SeTra'06. Volume 3., Electronic Communications of EASST (2006)

5. Hoffmann, B., Janssens, D., Eetvelde, N.V.: Cloning and expanding graph transformation rules for refactoring. ENTCS **152** (2006) 53–67
6. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and Systems Modeling **6**(3) (September 2007) 269–285
7. van Kempen, M., Chaudron, M., Kourie, D., Boake, A.: Towards proving preservation of behaviour of refactoring of UML models. In: SAICSIT '05, South African Institute for Computer Scientists and Information Technologists (2005) 252–259
8. Pérez, J., Crespo, Y.: Exploring a method to detect behaviour-preserving evolution using graph transformation. In: Proceedings of the Third International ERCIM Workshop on Software Evolution, ERCIM (2007) 114–122
9. Narayanan, A., Karsai, G.: Towards verifying model transformations. In Bruni, R., Varró, D., eds.: Proc. of GT-VMT '06. ENTCS, Vienna (2006) 185–194
10. Van Gorp, P., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In Stevens, P., Whittle, J., Booch, G., eds.: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 144–158
11. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In Fiadeiro, J.L., Inverardi, P., eds.: FASE'08. Volume 4961 of LNCS., Springer (2008) 347–361
12. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting. In: Proc. of FoSSaCS '04. Volume 2987 of LNCS. (2004) pp. 151–166
13. Hirschkoff, D.: Bisimulation verification using the up-to techniques. International Journal on Software Tools for Technology Transfer **3**(3) (August 2001) pp. 271–285
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series). Springer-Verlag New York, Inc., Secaucus, NJ, USA (2006)
15. Lambers, L.: Adhesive high-level replacement system with negative application conditions. Technical report, TU Berlin (2007)
16. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. Technical Report 12/08, TU Berlin (2008)
17. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. Mathematical Structures in Computer Science **16**(6) (2006) pp. 1133–1163
18. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Loewe, M.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations, World Scientific (1997) pp. 163–246
19. Rangel, G.: Bisimulation Verification for Graph Transformation Systems with Borrowed Contexts. PhD thesis, TU Berlin (2008) To appear.
20. Rangel, G., König, B., Ehrig, H.: Deriving bisimulation congruences in the presence of negative application conditions. In: Proc. of FOSSACS '08. Volume 4962 of LNCS., Springer (2008) 413–427