# Showing Full Semantics Preservation in Model Transformation – A Comparison of Techniques*

Mathias Hülsbusch[1], Barbara König[1], Arend Rensink[2], Maria Semenyak[3], Christian Soltenborn[3], and Heike Wehrheim[3]

[1] Abteilung für Informatik und Angewandte Kognitionswissenschaft, Universität Duisburg-Essen, Germany
[2] Department of Computer Science, University of Twente, The Netherlands
[3] Institut für Informatik, Universität Paderborn, Germany

**Abstract.** Model transformation is a prime technique in modern, model-driven software design. One of the most challenging issues is to show that the semantics of the models is not affected by the transformation. So far, there is hardly any research into this issue, in particular in those cases where the source and target languages are different.

In this paper, we are using two different state-of-the-art proof techniques (explicit bisimulation construction versus borrowed contexts) to show bisimilarity preservation of a given model transformation between two simple (self-defined) languages, both of which are equipped with a graph transformation-based operational semantics. The contrast between these proof techniques is interesting because they are based on different model transformation strategies: triple graph grammars versus in situ transformation. We proceed to compare the proofs and discuss scalability to a more realistic setting.

## 1 Background

One of today's most promising approaches for building complex software systems is the Object Management Group's *Model Driven Architecture* (MDA). The core idea of MDA is to first model the target system in an abstract, platform-independent way, and then to refine that model step by step, finally producing platform-specific, executable code. The refinement steps are to be performed automatically using so-called *model transformations*; the knowledge needed for each refinement step is contained in the respective transformation.

As a consequence, in addition to the source model's correctness, the correctness of the model transformations is crucial for MDA; if they contain errors, the target system might be seriously flawed. But how to ensure the correctness of a model transformation? In this paper, we take a formal approach: We want to *prove* that the presented model transformation is *semantics preserving*, i.e., we prove that the behaviour of source and generated target model is equivalent (in a very strict sense, discussed below) for *every* source model we potentially start with.

As an example of a realistically sized case for which behavioural preservation is desirable, in [5] we have presented a model transformation from UML Activity Diagrams

* Partially supported by DFG project Behaviour-GT.

[19] (called AD below) to TAAL [11], a Java-like textual language. The choice of this case is motivated by two reasons:

- It involves a transformation from an abstract visual language into a more concrete textual one, and hence it perfectly fits into the MDA philosophy.
- The semantics of both the source and target language (AD and TAAL) have been formally specified by means of graph transformation systems ([8] and [11], resp.).

The latter means that every AD model and every TAAL program give rise to a *transition system* modelling its execution. This in turn allows the application of standard concepts from concurrency theory in order to compare the executions and to decide whether they are indeed equivalent or not. Our aim is eventually to show *weak bisimilarity* between the transition system of any Activity Diagram and that of the TAAL program resulting from its transformation. Since weak bisimilarity is one of the most discriminating notions of behavioural equivalence (essentially preserving all properties in any reasonable temporal logic), we call this *full* semantic preservation.

Unfortunately, the size and complexity of the above problem are such that we have decided to first develop proof strategies for the intended result on a much more simplified version of the languages. In the current paper, we therefore apply the same question to two toy languages, inspired by AD and TAAL. Especially we model one non-trivial aspect: the token offer-based semantics of AD. Then, we solve the problem using two contrasting proof strategies.

The contribution of this paper lies in developing these two general strategies, carrying out the proofs for our example and afterwards comparing the strategies. Although simple, our example exhibits general characteristics of complex model-to-model transformations: different source and target languages, different levels of granularity of operational steps in the semantics and different labellings of steps. Our two proof strategies represent general approaches to proving semantics preservation of such model transformations.

The *first strategy* relies on a triple graph grammar-based definition of the model transformation (see [12,24]). Based on the resulting (static) triple graphs, we define an explicit bisimulation relation between the dynamic, run-time state graphs.

The *second strategy* relies instead on an in-situ definition of the model transformation and an extension of the operational semantics to the intermediate (hybrid) models. Using the theory of borrowed contexts (see [4]), we show that each individual model transformation step preserves the semantics.

The rest of the paper is structured as follows: Section 2 sets up a formal basis for the paper. Additionally, the source and target language and their respective semantics are introduced. Sect. 3 defines the model transformation, in both variants (triple graph grammar-based and in-situ). The actual proofs are worked out in Sections 4 and 5, respectively. Finally, Sect. 6 discusses and evaluates the results. Detailed proofs and additional information are contained in the extended version of this paper [10].

## 2 Definitions

### 2.1 Graphs and Morphisms

**Definition 1 (Graph).** *A graph is a tuple $G = \langle V, E, src, tgt, lab \rangle$, where $V$ is a finite set of nodes, $E$ a finite set of edges, $src, tgt\colon E \rightarrow V$ are source and target functions* *associating nodes with every edge, and* $lab\colon E \to \mathsf{Lab}$ *is an edge labelling function. We always assume* $V \cap E = \emptyset$.

For a given graph $G$, we use $V_G, E_G$ etc. to denote its components. Note that there is a straightforward (component-wise) definition of union and intersection over graphs, with the caveat that these operators may be undefined if the source, target or labelling functions are inconsistent.

In example graphs, we use the convention that self-edges may be displayed through node labels. That is, every node label in a figure actually represents an edge from that node to itself, with the given label. We now define morphisms as structure-preserving maps between graphs.

**Definition 2 (Morphism).** *Given two graphs* $G, H$*, a morphism* $f\colon G \to H$ *is a pair of functions* $(f_V\colon V_G \to V_H, f_E\colon E_G \to E_H)$ *from the nodes and edges of* $G$ *to those of* $H$ *which are consistent with respect to the source and target functions of* $G$ *and* $H$ *in the sense that* $src_H \circ f_E = f_V \circ src_G$, $tgt_H \circ f_E = f_V \circ tgt_G$ *and* $lab_H \circ f_E = lab_G$. *If both* $f_V$ *and* $f_E$ *are injective (bijective), we call* $f$ *injective (bijective).*

A bijective morphism is often called an *isomorphism*: if there exists an isomorphism from $G$ to $H$, we call them *isomorphic*. A frequently used notion of graph structuring is obtained by *typing* graphs over a fixed *type graph*.

**Definition 3 (Typing).** *Given two graphs* $G, T$*, the graph* $G$ *is said to be* typable over $T$ *if there exists a* typing morphism $t\colon G \to T$. *A* typed graph *is a graph* $G$ *together with such a typing morphism, say* $t_G$. *Given two graphs* $G, H$ *typed over the same type graph (using typing morphisms* $t_G$ *and* $t_H$*), a* typed graph morphism $f\colon G \to H$ *is a morphism that preserves the typing, i.e., such that* $t_G = t_H \circ f$.

Besides imposing some structural constraints over graphs, typing also provides an easy way to restrict to subgraphs:

**Definition 4 (Type restriction).** *Let* $T, U$ *be graphs such that* $U \subseteq T$*, and let* $G$ *be an arbitrary graph typed over* $T$ *via* $t : G \to T$. *The* restriction of $G$ to $U$*, denoted* $\pi_U(G)$*, is defined as the graph* $H$ *such that*

– $V_H = \{v \in V_G \mid t(v) \in V_U\}$, $E_H = \{e \in E_G \mid t(e) \in E_U\}$,
– $src_H = src_G \restriction_{E_H}$, $tgt_H = tgt_G \restriction_{E_H}$ *and* $lab_H = lab_G \restriction_{E_H}$.

The set of graphs with their morphisms form a category, which we will denote by $\mathsf{Graph}$.

## 2.2 Graph Languages

In this paper we consider model transformation between two languages. In particular, we consider *graph languages*, i.e. sets of graphs; the models are the graphs themselves. We concentrate on a running example where there are two distinct, very simple graph languages denoted $\mathcal{A}$ and $\mathcal{B}$. Fig. 1 shows type graphs for the languages, denoted $T_{\mathcal{A}}^{\mathsf{st}}$ and $T_{\mathcal{B}}^{\mathsf{st}}$, respectively. They describe the typing of the *static* parts of our two languages. We will sometimes also call these the (static) metamodels of the two languages. The

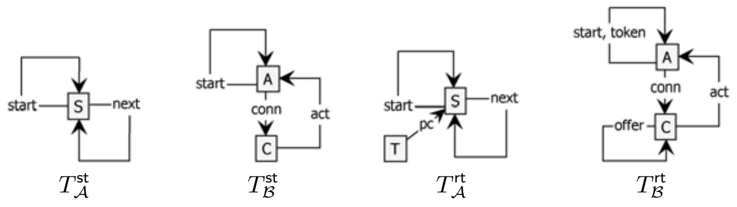
start
S
next
start
A
conn
act
C
start
S
next
pc
T
start, token
A
conn
act
offer
C
$T^{\mathsf{st}}_{\mathcal{A}}$
$T^{\mathsf{st}}_{\mathcal{B}}$
$T^{\mathsf{rt}}_{\mathcal{A}}$
$T^{\mathsf{rt}}_{\mathcal{B}}$

**Fig. 1.** Static ($\mathsf{st}$) and run-time ($\mathsf{rt}$) type graphs for graph languages $\mathcal{A}$ and $\mathcal{B}$

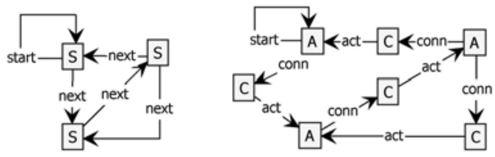

**Fig. 2.** Example graphs of languages $\mathcal{A}$ (left) and $\mathcal{B}$ (right)

figure also shows the corresponding extended *run-time* type graphs, which will be discussed below (Section 2.4).

The type graphs themselves impose only weak structure: not all graphs that can be typed over the $\mathcal{A}$- and $\mathcal{B}$-type graphs are considered to be part of the languages. Instead, we impose the following further constraints on the static structure:

*Language* $\mathcal{A}$ consists of next-connected S-labelled nodes (*statements*). There should be a single S-node with a start-edge to itself, from which all other nodes are reachable (via paths of next-edges). Furthermore no next-loops are allowed.

*Language* $\mathcal{B}$ consists of bipartite graphs of A- (*action*) and C-labelled (*connector*) nodes. Every C-node has exactly one incoming conn-edge and exactly one outgoing act-edge; the opposite nodes of those edges must be distinct. Like $\mathcal{A}$-graphs, $\mathcal{B}$-graphs have exactly one node with a start-self-edge, from which all other nodes are reachable (via paths of conn- and act-edges).

Small example graphs are shown in Fig. 2. We use $\mathcal{G}^{\mathsf{st}}_{\mathcal{A}}$ ($\mathcal{G}^{\mathsf{st}}_{\mathcal{B}}$) to denote the set of all well-formed (static) $\mathcal{A}$-graphs ($\mathcal{B}$-graphs).

## 2.3 Rules and Rule Systems

To specify the semantics of our languages, we have to formally describe changes on our graphs. This is done by means of *graph transformation rules*. A rule describes the change of (parts of) a graph by means of a before and after template (the left-hand and right-hand hand side of a rule); the interface fixes the part on which left and right hand side have to agree.

**Definition 5 (Transformation rule).** *A graph transformation rule is a tuple $r = \langle L, I, R, \mathcal{N} \rangle$, consisting of a left hand side (LHS) graph $L$, an interface graph $I$, a right hand side (RHS) graph $R$, and a set $\mathcal{N} \subseteq$* Graph *of* negative application conditions (NAC's)*, which are such that $L \subseteq N$ for all $N \in \mathcal{N}$. The interface $I$ is the intersection of $L$ and $R$ ($I = L \cap R$).*

We let $\mathsf{Rule}$ denote the set of rules. A rule (without a NAC) is basically a pair of injective morphisms in $\mathsf{Graph}$: $L \leftarrow I \rightarrow R$. The diagram for a rule with NACs is this basic span together with injective morphisms from $L$ to the elements of $\mathcal{N}$. For a single NAC $N$, a rule has the following form: $N \leftarrow L \leftarrow I \rightarrow R$. There are other definitions of graph transformation rules in the literature, the one used here is the one for double-pushout rewriting (DPO-rewriting).

A transformation rule $r = \langle L, I, R, \mathcal{N} \rangle$ is *applicable* to a graph $G$ (called the *host graph*) if there exists an injective *match* $m\colon L \to G$ such that for no $N \in \mathcal{N}$ there exists a match $n\colon N \to G$ with $m = n \restriction_L$ (i.e., all negative application conditions are *satisfied*), and moreover, the *dangling edge condition* holds: for all $e \in E_G$, $src(e) \in m(V_L \setminus V_I)$ or $tgt(e) \in m(V_L \setminus V_I)$ implies $e \in m(E_L \setminus V_I)$. This condition can be understood by realising that the elements of $G$ that are in $m(L)$, but not in $m(I)$, are scheduled to be deleted by the rule, whereas the elements in $m(I)$ are preserved (see below). Hence we can not delete a node without explicitly deleting all adjacent edges.

Given such a match $m$, the *application* of $r$ to $G$ is defined by extending $m$ to $L \cup R$, by choosing distinct "fresh" nodes and edges (outside $V_G$ and $E_G$, respectively) as images for $V_R \setminus V_L$ and $E_R \setminus E_L$ and adding those to $G$. This extension results in a morphism $\bar{m}\colon (L \cup R) \to C$ for some extended graph $C \supseteq G$. Now let $H$ be given by $V_H = V_C \setminus m(V_L \setminus V_R)$, $E_H = E_C \setminus m(E_L \setminus E_R)$, together with the obvious restriction of $src_C$, $tgt_C$ and $lab_C$ to $E_H$. The graph $H$ is called the *target* of the rule application; we write $G \xrightarrow{r,m} H$ to denote that $m$ is a valid match on host graph $G$, giving rise to target graph $H$, and $G \xrightarrow{r} H$ to denote that there is a match $m$ such that $G \xrightarrow{r,m} H$. Note that $H$ is not uniquely defined, due to the freedom in choosing the fresh images for $V_R \setminus V_L$ and $E_R \setminus E_L$; however, it is well-defined up to isomorphism.

**Definition 6 (Rule system).** *A rule system is a partial mapping $\mathcal{R}\colon \mathsf{Sym} \rightharpoonup \mathsf{Rule}$. Here,* $\mathsf{Sym}$ *is a universe of* rule names.

## 2.4 Language Semantics

In the context of the two languages defined in Section 2.2, we can use graph transformation rules for two separate purposes: to give a grammar that precisely and formally defines the languages or to specify the operational language semantics. In the latter case, the transformation rules describe patterns of state changes.

We will demonstrate the second usage here, by giving operational rules for $\mathcal{A}$-graphs and $\mathcal{B}$-graphs. This means that the graphs will represent run-time states. As we will see, this will involve auxiliary node and edge types that do not occur in the language type graphs. Fig. 1 shows extended type graphs $T^{\mathsf{rt}}_{\mathcal{A}}$ and $T^{\mathsf{rt}}_{\mathcal{B}}$ that include these *run-time* types. For $\mathcal{A}$, a $\mathsf{T}$-node (of which there can be at most one) models a *thread*, through a single program counter ($\mathsf{pc}$-labelled edge). For $\mathcal{B}$, we use $\mathsf{token}$- and $\mathsf{offer}$-loops which play a similar role; details will become clear below. Similar to the static part, we use $\mathcal{G}^{\mathsf{rt}}_{A}$ ($\mathcal{G}^{\mathsf{rt}}_{B}$) to denote the set of well-formed (run-time) $\mathcal{A}$-graphs ($\mathcal{B}$-graphs). The semantics of $\mathcal{A}$- and $\mathcal{B}$-models is defined in Fig. 3. Note that the figure shows the rules in DPO style, i.e. the middle part gives the interface $I$, and the sides are $L$ and $R$, given as $L \leftarrow I \rightarrow R$. Additionally, NACs might be present.

We let $dom(\mathcal{R}_{\mathcal{A}}) = \{\mathsf{initA}, \mathsf{movePC}\}$ and $dom(\mathcal{R}_{\mathcal{B}}) = \{\mathsf{initB}, \mathsf{createO}, \mathsf{moveT}\}$ be the names in the rule systems for the $\mathcal{A}$- and $\mathcal{B}$-models, the mapping to rules follows

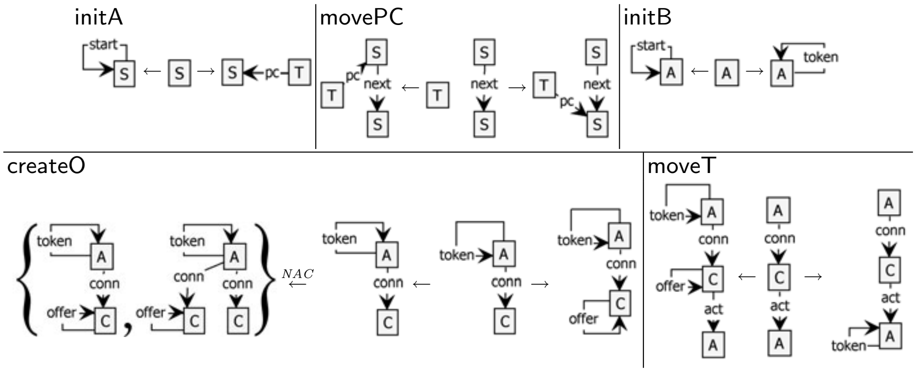

**Fig. 3.** Operational rules for $\mathcal{A}$ (initA and movePC) and $\mathcal{B}$ (initB, createO and moveT)

Fig. 3. Intuitively, the init-rules perform an initialisation of the run-time system, setting the program counter to the start statement (in $\mathcal{A}$) or putting a token onto a start action (in $\mathcal{B}$). Rule movePC simply moves the program counter to the next statement, createO moves an offer to a C-node and moveT moves the token. The semantics of $\mathcal{A}$- and $\mathcal{B}$-graphs is completely fixed by these rules, giving rise to a labelled transition system summarizing all these executions.

**Definition 7 (Labelled transition system).** *An $L$-labelled* transition system *(LTS) is a structure $S = \langle Q, \rightarrow, \iota\rangle$, where $Q$ is a set of states and $\rightarrow \subseteq Q \times L \times Q$ is a set of transitions labelled over some set of labels L. Furthermore $\iota \in Q$ is the start state.*

In our case, the states are graphs and the transitions are rule applications. That is, given a rule system $\mathcal{R}$ and a start graph $G$, we obtain a $dom(\mathcal{R})$-labelled transition system by recursively applying all rules to all graphs. We will denote this transition system by $S(G)$ (leaving the rule system $\mathcal{R}$ implicit). For instance, the LTS of an $\mathcal{A}$-graph $G$ is $S(G) = (\mathcal{G}^{\mathrm{rt}}_{\mathcal{A}}, \rightarrow_{\mathcal{A}}, G)$, where $\rightarrow_{\mathcal{A}}$ is defined by the rules in $\mathcal{R}_A$.

Semantic equivalence comes down to equivalence of the LTSs generated by two different graphs. There are several notions of equivalence over LTSs; see, e.g., [25]. In this paper, we use *weak bisimulation*. Weak bisimulation requires two states to mutually simulate each other, where a simulation may however involve internal (unobservable) steps. As usual, we use the special transition label $\tau$ to denote such internal steps.

For states $q, q' \in Q$ and a label $\alpha$, we write $q \overset{\alpha}{\Longrightarrow} q'$ if $q \xrightarrow{\tau}{}^{*} \xrightarrow{\alpha} \xrightarrow{\tau}{}^{*} q'$ and use $\overset{\varepsilon}{\Longrightarrow}$ to stand for $\xrightarrow{\tau}{}^{*}$. Furthermore, we define for (visible or invisible) labels $\alpha$ the following function$\widehat{\ }$: $\widehat{\tau} = \epsilon$ and $\widehat{\alpha} = \alpha$ if $\alpha \neq \tau$.

**Definition 8 (Weak bisimilarity).** *Weak bisimilarity between two labelled transition systems $S_1, S_2$ is a relation $\approx \subseteq Q_1 \times Q_2$ such that whenever $q_1 \approx q_2$*

- *If $q_1 \xrightarrow{\alpha} q_1'$, then $q_2 \overset{\widehat{\alpha}}{\Longrightarrow} q_2'$ such that $q_1' \approx q_2'$;*
- *If $q_2 \xrightarrow{\alpha} q_2'$, then $q_1 \overset{\widehat{\alpha}}{\Longrightarrow} q_1'$ such that $q_1' \approx q_2'$.*

*We call $S_1$ and $S_2$ as a whole weakly bisimilar, denoted $S_1 \approx S_2$, if there exists a weak bisimilarity relation between $S_1$ and $S_2$ such that $\iota_1 \approx \iota_2$.*

## 2.5 Semantics-Preserving Model Transformation

Our objective is to compare the LTSs of graphs of languages $\mathcal{A}$ and $\mathcal{B}$. In Section 3 we will define a (relational) model transformation $MT \subseteq \mathcal{G}^{\mathsf{st}}_{\mathcal{A}} \times \mathcal{G}^{\mathsf{st}}_{\mathcal{B}}$ translating $\mathcal{A}$-graphs to $\mathcal{B}$-graphs. We aim at proving this model transformation to be *semantics preserving*, in the sense that the LTSs of source and target models are always weakly bisimilar.

However, there is an obvious problem: the LTSs of $\mathcal{A}$- and $\mathcal{B}$-graphs do not have the same labels, in fact $dom(\mathcal{R}_{\mathcal{A}}) \cap dom(\mathcal{R}_{\mathcal{B}}) = \emptyset$. Nevertheless, there is a clear intuition which rules correspond to each other: on the one hand the two initialisation rules, and on the other hand the rules movePC and createO. The reason for taking the latter two as corresponding is that both rules decide on where control is moving. The rule moveT has no matching counterpart in the $\mathcal{A}$-language, it can be seen as an *internal* step of the $\mathcal{B}$-language, completing a step initiated by createO. These observations give rise to the following renaming of the labels (i.e., the rule names) to a common set of names.

$$\begin{array}{llll} map_{\mathcal{A}}: & \mathsf{initA} \mapsto \mathsf{init}, & \mathsf{movePC} \mapsto \mathsf{move} & \\ map_{\mathcal{B}}: & \mathsf{initB} \mapsto \mathsf{init}, & \mathsf{createO} \mapsto \mathsf{move}, & \mathsf{moveT} \mapsto \tau \end{array}$$

We call such a mapping $map\colon dom(\mathcal{R}) \to \mathsf{Sym}$ (for a given rule system $\mathcal{R}$) *non-trivial* if it does not map every rule name to $\tau$.

**Definition 9 (Preservation of semantics).** *Given two (graph) languages $\mathcal{G}^{st}_{\mathcal{A}}, \mathcal{G}^{st}_{\mathcal{B}}$, a model transformation $MT \subseteq \mathcal{G}^{st}_{\mathcal{A}} \times \mathcal{G}^{st}_{\mathcal{B}}$ is* semantics-preserving *if there are non-trivial mapping functions $map_{\mathcal{A}}\colon dom(\mathcal{R}_{A}) \to \mathsf{Sym}$, $map_{\mathcal{B}}\colon dom(\mathcal{R}_{B}) \to \mathsf{Sym}$ such that for all $G_A \in \mathcal{G}^{st}_{\mathcal{A}}$, $G_B \in \mathcal{G}^{st}_{\mathcal{B}}$ with $MT(G_A, G_B)$*

$$map_{\mathcal{A}}(S(G_A)) \approx map_{\mathcal{B}}(S(G_B)) \ .$$

# 3 Model Transformation

Our model transformation needs to translate $\mathcal{A}$-models into $\mathcal{B}$-models. We will actually present two definitions of the transformation, both tailored towards the specific proof technique used for showing semantics preservation.

## 3.1 Triple Graph Grammars

Our first transformation uses triple graph grammars (TGGs). TGG rules [24,12] typically capture transformations between models of *different* types. Triple graphs can be separated into three subgraphs, typed over their own type graphs. Two of these subgraphs evolve simultaneously while the third keeps correspondences between them. For our example, we have the two type graphs $T^{\mathsf{rt}}_{\mathcal{A}}$ and $T^{\mathsf{rt}}_{\mathcal{B}}$ which — for forming a type graph for TGGs — are conjoined and augmented with one new correspondence G-node (the glue); see Fig. 4, resulting in a combined type graph $T^{\mathsf{rt}}_{\mathcal{AB}}$.

Normally, for a transformation, the source model is given in the beginning and is then gradually transformed. TGG rules however build two models simultaneously, matching each part of the source model to the target one. This allows to keep correspondences between transformed elements and to prove certain properties of the corresponding graphs. The TGG rules for the $\mathcal{A}$ to $\mathcal{B}$ transformation are given in Fig. 5.

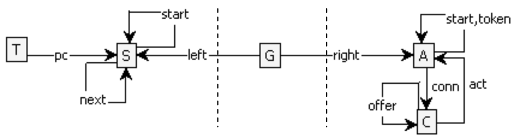

**Fig. 4.** Type graph $T^{\mathsf{rt}}_{\mathcal{AB}}$ for TGG graph rules

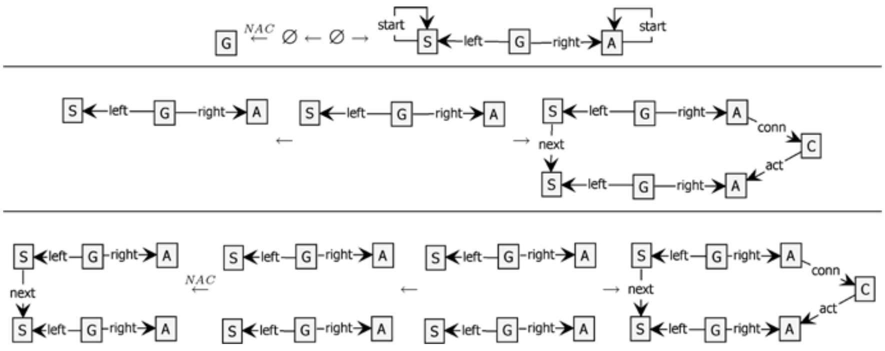

**Fig. 5.** TGG transformation rules

These rules incrementally build combined $\mathcal{A}$ and $\mathcal{B}$-graphs. Initially, only the upper rule in Fig. 5 can be applied; it constructs one S- and one A-node connected via one correspondence G-node. The middle rule creates further S-, A- and C-nodes together with their correspondences; the lower rule simultaneously generates next-edges between S-nodes and connections via C-nodes between *corresponding* A-nodes. Let $\mathcal{G}^{\mathsf{rt}}_{AB}$ denote the set of graphs obtained by applying the three TGG rules on an empty start graph. To obtain the actual translation, restrict $\mathcal{G}^{\mathsf{rt}}_{AB}$ to the type graphs of $\mathcal{A}$ and $\mathcal{B}$. Using the definition of type restriction as given in Section 2, the model transformation $MT$ thus works as follows: Given an $\mathcal{A}$-graph $G_A$ and a $\mathcal{B}$-graph $G_B$, we have $MT(G_A, G_B)$ exactly if there is some $G_{AB} \in \mathcal{G}^{\mathsf{rt}}_{AB}$ such that $G_A = \pi_{T^{\mathsf{st}}_{\mathcal{A}}}(G_{AB})$ and $G_B = \pi_{T^{\mathsf{st}}_{\mathcal{B}}}(G_{AB})$.

## 3.2 In-Situ Transformation

Instead of building two models simultaneously, in-situ transformations destroy the source model while building the target model. This has the disadvantage of leading to "mixed" states, with components of both the source and the target model. This necessitates additional operational rules (see Section 5). On the other hand, in-situ transformations describe a clear evolution process. This is better suited as a basis for proof strategy 2, which relies on a congruence result for bisimilarity: the basic idea is that replacing a part of the model does not affect behavioural equivalence of the entire model.

We will now present the in-situ transformation rules, which are shown in Fig. 6. The first rule relabels nodes by replacing the label S by the label A[1]. The second rule replaces a next-edge by a connection via a C-node. The third rule replaces the program

[1] Remember that labels are represented by loops on an unlabelled node.

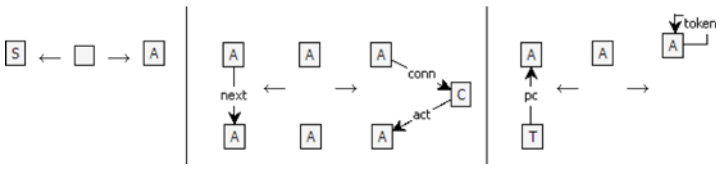

**Fig. 6.** In-situ transformation rules from language $\mathcal{A}$ to language $\mathcal{B}$

counter by a token and allows the transformation of run-time models. We have reached a model in language $\mathcal{B}$ as soon as no further rule applications are possible. We define that $MT(G_A, G_B)$ iff $G_A$ is transformed into $G_B$ via the rules in Fig. 6.

## 3.3 Comparison

In this section we argue that both strategies define the same model transformation. Assume that a graph $G_A$ is transformed into a graph $G_B$ via the TGG transformation of Section 3.1. This means that $G_A$ and $G_B$ are constructed simultaneously by the TGG grammar and arise as projections of a graph $G_{AB}$. Then we can apply the in-situ rules of Fig. 6 to $G_A$, obtaining the corresponding items of $G_B$.

The other direction is slightly more complicated. Assume that we are given a graph $G_A$ of language $\mathcal{A}$. Then, with the TGG rules, we generate a graph $G_{AB}$ which projects (via $\pi_{T^{\mathrm{st}}_{\mathcal{A}}}$) to $G_A$. We can then show, by induction on the length of this generating sequence and by using the fact that the transformation rules are confluent, that the graph $\pi_{T^{\mathrm{st}}_{\mathcal{B}}}(G_{AB})$ obtained in this way coincides with $G_B$, the graph generated by applying the in-situ transformation rules as long as possible.

## 4 Proof Strategy 1

In this section, we present our first approach to proving semantic preservation of the model transformation on all source models (for more details see the extended version [10]). This proof strategy uses the correspondences generated by the TGG rules, despite the fact that the semantic rules are applied on the individual models, based on the following two observations.

**First observation:** Both for $\mathcal{A}$ and $\mathcal{B}$-models, the operational rules keep the syntactic, static structure of a model, except for start-edges: all S-nodes and next-edges, and all A, C-nodes and conn, act-edges stay the same.

To formulate structural correspondences, we introduce the following notation. For an S-node $v_S$ and an A-node $v_A$, we write $corr(v_S, v_A)$ if there is a G-node $v_G$ and a left-edge from $v_G$ to $v_S$ and a right-edge from $v_G$ to $v_A$. For an edge $e$ labelled label going from a node $v$ to $v'$, we simply write label$(v, v')$. We also use these as predicates. The first result shows that correspondences between S and A-nodes are unique. Here, $\exists!$ stands for "there exists exactly one".

**Proposition 10.** *Let $G \in \mathcal{G}^{rt}_{AB}$, $v_S$ an* S*-node and $v_A$ an* A*-node in $G$. Then the following two properties hold: (A) $\exists! v$ of type* A *such that $corr(v_S, v)$, and (B) $\exists! v$ of type* S *such that $corr(v, v_A)$.*

A number of further results show that (1) corresponding nodes either both or none have start-edges, and (2) next-edges between S-nodes will generate connections via C-nodes between corresponding A-nodes and vice versa.

**Second observation:** Correspondences between nodes in $\mathcal{A}$-models and $\mathcal{B}$-models are kept during application of semantic rules. Predicate *corr* as well as Prop. 10 and properties (1) and (2) can thus also be applied to separate $\mathcal{A}$ and $\mathcal{B}$-graphs.

**Theorem 11.** *Let $G^0_A$, $G^0_B$ be an $\mathcal{A}$- and a $\mathcal{B}$-graph such that $MT(G^0_A, G^0_B)$. Then*

$$map_A(S(G^0_A)) \approx map_B(S(G^0_B))$$

For the proof, we need to construct a weak bisimulation relation $\mathcal{R}$ (defining $\approx$) between the states of the first and the second LTS:

$$\mathcal{R} = \{(G_A, G_B) \in \mathcal{G}^{rt}_{\mathcal{A}} \times \mathcal{G}^{rt}_{\mathcal{B}} \mid \exists G_{AB} \in \mathcal{G}^{rt}_{\mathcal{AB}}$$

(1) $\pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{A}} \setminus \mathsf{start}}(G_A) = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{A}} \setminus \mathsf{start}}(G_{AB}) \wedge \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{B}} \setminus \mathsf{start}}(G_B) = \pi_{\mathsf{T}^{\mathsf{st}}_{\mathsf{B}} \setminus \mathsf{start}}(G_{AB})$,

(2) $\forall$ S-nodes $v_S$ in $G_A$, A-nodes $v_A$ in $G_B$ s.t. $corr(v_S, v_A)$:
start$(v_S)$ iff start$(v_A)$,

(3) $\forall$ S-nodes $v_S$ in $G_A$, A-nodes $v_A$ in $G_B$ s.t. $corr(v_S, v_A)$: $\exists v_T$ s.t. pc$(v_T, v_S)$
iff (i) token$(v_A) \wedge \forall v_C$ s.t. conn$(v_A, v_C)$ : $\neg$offer$(v_C)$ or
(ii) $\neg$token$(v_A) \wedge \exists v_C, v'_A$ : token$(v'_A) \wedge$ offer$(v_C) \wedge$
conn$(v'_A, v_c) \wedge$ act$(v_C, v_A)$,

(4) $\exists v_T, v_S$ : pc$(v_T, v_S) \iff \neg\exists v'_S$ : start$(v'_S) \wedge$
$\exists v_A$ : token$(v_A) \iff \neg\exists v'_A$ : start$(v'_A) \wedge$
$\neg\exists v_A$ : start$(v_A) \implies \exists! v'_A$ : token$(v'_A) \wedge$
$\forall v_C$ : offer$(v_C) \implies \exists v_A$ : token$(v_A) \wedge$ conn$(v_A, v_C) \wedge$
$\neg\exists v_S$ : start$(v_S) \implies \exists! v'_S$ s.t. $\exists v_T$ : pc$(v_T, v'_S)$ $\}$

It contains all pairs of $\mathcal{A}$ and $\mathcal{B}$-graphs which (1) in their static structure (except for start) still follow the structure generated by the TGG rules, (2) have start-edges only on corresponding nodes, (3) exhibit run-time properties only on corresponding nodes, and (4) obey certain well-formedness criteria for run-time elements.

Fig. 7 further illustrates condition (3). We have two possibilites for run-time elements in matching states: either the pc-edge is on an S-node and the token is on the corresponding A-node and no further offers exist (left), or the pc-edge is on a node for which the corresponding A-node has no token yet, but an offer has already been created and is ready to move the token to the A-node by means of the invisible step moveT

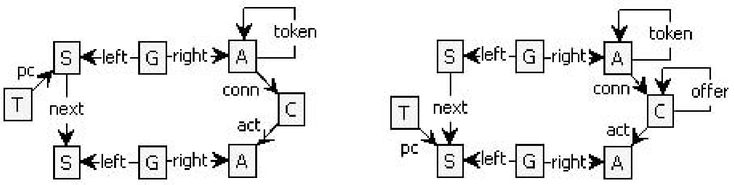

**Fig. 7.** Illustration of condition (3): Left (i), right (ii)

(right). We show that the relation $\mathcal{R}$ is a weak bisimulation by proving that the states of transition systems can mimic each others moves. Due to space limitations we cannot give the full proof here, which can instead be found in the extended version [10].

## 5 Proof Strategy 2

### 5.1 The Borrowed Context Technique

In the following we will describe a different proof strategy, based on the borrowed context technique [4,21], which refines a labelled transition system (or even unlabelled reaction rules) in such a way that the resulting bisimilarity is a congruence [14]. Weak bisimilarity as in Def. 8 is usually not a congruence. By a congruence we mean a relation over graphs that is preserved by contextualization, i.e., by gluing with a given environment graph over a specified interface. This is a mild generalization of standard graph rewriting in that we consider "open" graphs, equipped with a suitable interface.

The basic idea behind the borrowed context technique is to describe the possible interactions with the environment. In addition to existing labels, we add the following information to a transition: what is the (minimal) context that a graph with interface needs to evolve? More concretely we have transitions of the form

$$(J \to G) \xrightarrow{\alpha,(J \to F \leftarrow K),\mathcal{N}} (K \to H)$$

where the components have the following meaning: $(J \to G)$ is the original graph with interface $J$ (given by an injective morphism from $J$ to $G$) which evolves into a graph $H$ with interface $K$. The label is now composed of three entities: the original label $\alpha = map(r)$ stemming from the operational rule $r$ (as detailed in Section 2.5) and furthermore two injective morphisms $(J \to F \leftarrow K)$ detailing what is borrowed from the environment. The graph $F$ represents the additional graph structure, whereas $J, K$ are its inner and the outer interface. Finally we provide a set $\mathcal{N}$ of negative borrowed contexts, describing negative constraints on the environment (see also [21]). We are using a saturated and weak version of bisimulation (see the extended version [10]).

### 5.2 Using Borrowed Contexts for Verification of Model Transformation

For in-situ model transformation within the same language, applications of the borrowed context technique are straightforward: show for every transformation rule that the left-hand and right-hand sides $L, R$ with interface $I$ are bisimilar with respect to the operational rules. Then the source model must be bisimilar to the target model by the congruence result. This idea has been exploited in [22] for showing behaviour preservation of refactorings.

However, in order to apply the idea above in our situation it is necessary to have an operational semantics also for "mixed" (or hybrid) models which incorporate components of both the source and the target model. Hence below we introduce such a mixed operational semantics, which has to satisfy the following conditions: (i) the mixed rules are *not* applicable to a pure source or target model; (ii) it is possible to show (borrowed context) bisimilarity of left-hand and right-hand sides of all transformation rules. Finally, observe that our final aim is to show bisimilarity of closed graphs, i.e., of graphs with empty interface. It can be shown that if all left-hand sides are connected, the notion of bisimilarity induced by borrowed contexts coincides with the standard one.

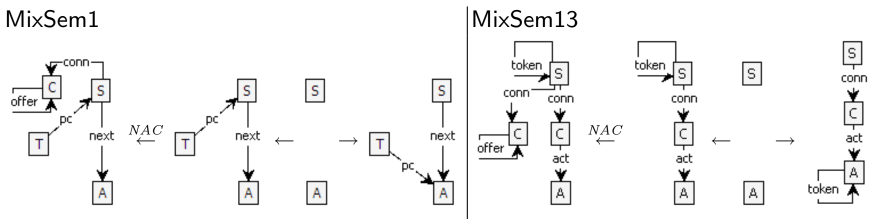

**Fig. 8.** Some rules of the operational semantics of mixed models

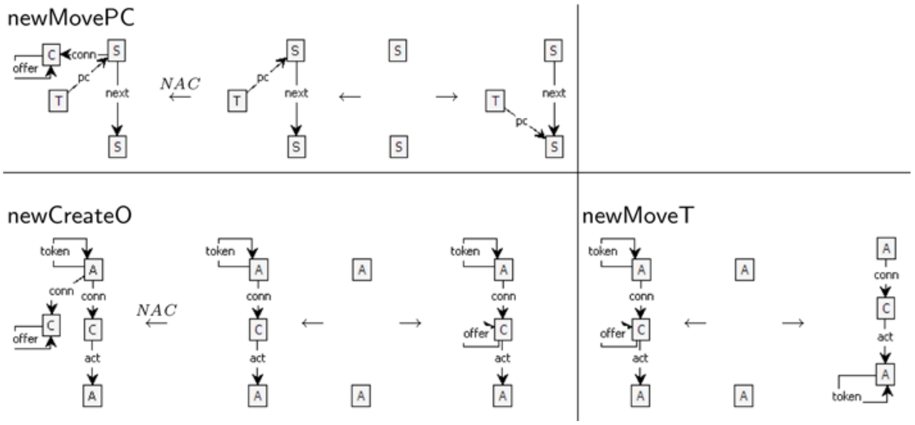

**Fig. 9.** Modified operational rules for the source and target languages

## 5.3 Rules of the Mixed Semantics

There are sixteen additional rules for the mixed semantics. Seven of them handle the behaviour of pc-edges at A-nodes, seven the semantics of the token-edge at an S-node and two of them are mixed counterparts to the initialization rules. Fig. 8 shows two examples of mixed rules, the rest are provided in [10]. Here we work with a single function $map$ (see Section 2.5), both rules in Fig. 8 are mapped to move.

Furthermore, we modify some of the operational rules of Fig. 3: first, we equip several rules, also of the source semantics (language $\mathcal{A}$) with NACs (without changing the operational behaviour). Second, we restrict to a minimal interface by deleting and recreating the connections (see Fig. 9). Due to the layout of the graphs, this does not modify the semantics. Both modifications are needed to make the proof work and the latter modification is also very convenient since it allows us to derive fewer labels.

## 5.4 The In-Situ Transformation Preserves Weak Bisimilarity

**Theorem 12.** *The left-hand sides and right-hand sides of the three in-situ transformation rules in Fig. 6 are weakly bisimular, with respect to the borrowed contexts technique, under the rules of the mixed semantics.*

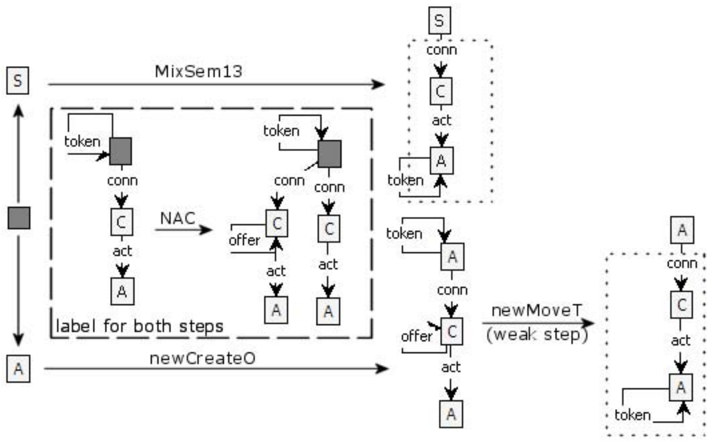

**Fig. 10.** Example of a label derivation using the borrowed context technique

*Since weak bisimilarity is a congruence [10] and borrowed context bisimilarity coincides with standard bisimilarity (see Def. 8) on source and target models, this implies that* $map(S(G_A)) \approx map(S(G_B))$ *whenever* $MT(G_A, G_B)$.

We give some intuition on the label derivation process by discussing one example, which needs the handling of weak moves and NACs (see Fig. 10).

In the labelled transition system, the graph consisting only of an S-node makes a move (with rule MixSem13) with the label shown in the (big) dashed box, i.e., it borrows a token, a C-node and an A-node. Spelling out the transition labels more concretely we have $\alpha = \mathsf{move}$, $F$ is the graph in the dashed box on the left (where the grey node represents both interfaces $J, K$) and the only NAC in $\mathcal{N}$ is given on the right. The corresponding graph (the A-node) can answer this step with the same label, by making a step with rule newCreateO plus a weak step ($\tau$) with rule newMoveT. After this second step, using an up-to-context proof technique, the same context (see dotted boxes) can be removed from both graphs, leaving the original pair of graphs already in the relation.

On the other hand, the answer to the newCreateO-step is with rule MixSem13. So the pair of graphs reached after one step has to be in the bisimulation as well and we have to check that they can mimic each others moves.

The entire bisimulation relation only contains five pairs, three are the in-situ transformation rules of Fig. 6 and two additional ones are needed. However, it is necessary to derive a large number of labels to prove that it is a bisimulation.

## 6 Discussion and Evaluation

Providing proof techniques for showing that full semantics preservation of a set of model transformation rules, for arbitrary source models, is a very difficult problem, on which there has been little work so far. The difficulty of the problem stems from three aspects: first, we need to show bisimilarity in transition systems based on graphs, a topic that has only recently started to receive attention; second, we do not only have to prove bisimilarity for a given pair of start graphs, but for an infinite set of pairs of source and corresponding target models; and third, we want to address this on the level of a reusable proof *technique*, and not just a single proof for a given model transformation.

We feel that so far little progress has been made in tackling the inherent underlying difficulty. Hence it is our strong feeling that it is first necessary to consider case studies of modest size to clearly outline and evaluate possible solutions.

The case study that was chosen for this paper might seem small, but it already incorporates several non-trivial aspects: a heterogeneous setup (different source and target languages), negative application conditions and the need for weak bisimilarity, since one step in the source model has to be matched by two steps in the target model.

Our two proof strategies reflect two major possibilities: either to work out a direct proof manually – which could be verified with a theorem prover – or to use a semi-automatic method based on bisimulation proof theory.

*Direct approach.* The direct bisimulation proof based on triple graph grammars uses little additional theory and can be carried out by resorting to standard proof methodology. Because of that it is more flexible than the borrowed context technique and can deal with the rules of the operational semantics without modification.

*Borrowed contexts.* Here we extended the borrowed context technique to work with weak bisimilarity, which is a novel contribution. The technique seems to be easier to mechanize than the direct proof: the label derivation process can be done fully automatically and, at least in the case where a finite bisimulation up-to context exists, there are possibilities to find it via an algorithm as suggested in [9]. We also have some initial ideas for automatically generating the mixed semantics (by applying the transformation rules to the left-hand sides of the operational rules).

*Summary.* We were able to make both proofs work with a reasonable effort, but further work is necessary in order to make the approach scale. We conclude that additional techniques, in particular mechanisation, will be needed to address realistic languages such as the ones in [5]. However, we do see a lot of unused potential in exploiting bisimulation theory and congruence results as was done in the second proof strategy. In the future it will also be interesting to study refactoring cases, rather than transformations between distinct modelling languages: they promise to be easier, because they involve only a single operational semantics. Furthermore we have to consider whether weak bisimilarity is the appropriate behavioural equivalence in all instances.

*Related work.* The work closest to ours in its objective of showing semantic preservation for a transformation between models of different types is [6]. They present a mechanised proof of semantics preservation (wrt. some version of bisimilarity — the paper does not contain an explicit definition) for a transformation of automata to PLC-code, based on TGG rules. This proof faced some problems since it was not trivial to present graph transformation within Isabelle/HOL.

Although there is extensive work on the verification of model transformations, to our knowledge there are only few attempts to show that transformations will always transform source models into behaviourally equivalent target models. For instance, [1] discusses several proof techniques (bisimulation, model-checking), but does not really explain how they could be exploited to prove full semantics preservation.

As opposed to general model transformation, there has been more work on showing correctness of refactorings. The methods presented in [26,20,17,7] address behaviour preservation in model refactoring, but are in general limited to checking a certain number of models. The employment of a congruence result is also proposed in [3] which uses the process algebra CSP as a semantic domain. The techniques used in [15,23] mainly treat state-based models, using set theory and predicate logic to show equivalences. In [2] it is shown how to exploit confluence results for graph transformation systems in order to show correctness of refactorings. A number of approaches also focus on preserving specific aspects instead of the full semantics (see [16]).

Instead of generally proving correctness of a transformation, a number of approaches, also in the area of compiler validation, carry out run-time checks of equivalence between a given source and generated target model [17,18,13].

## References

1. Barbosa, P.E.S., Ramalho, F., de Figueiredo, J.C.A., dos, S., Junior, A.D., Costa, A., Gomes, L.: Checking semantics equivalence of MDA transformations in concurrent systems. The Journal of Universal Computer Science 11, 2196–2224 (2009)
2. Baresi, L., Ehrig, K., Heckel, R.: Verification of model transformations: A case study with BPEL. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 183–199. Springer, Heidelberg (2007)
3. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings by rule extraction. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 347–361. Springer, Heidelberg (2008)
4. Ehrig, H., König, B.: Deriving bisimulation congruences in the DPO approach to graph rewriting with borrowed contexts. MSCS 16(6), 1133–1163 (2006)
5. Engels, G., Kleppe, A., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 94–109. Springer, Heidelberg (2008)
6. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards verified model transformations. In: Workshop on Model Development, Validation and Verification, pp. 78–93 (2006)
7. Gorp, P.V., Stenten, H., Mens, T., Demeyer, S.: Towards automating source-consistent UML refactorings. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 144–158. Springer, Heidelberg (2003)
8. Hausmann, J.: Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Languages. PhD thesis, University of Paderborn (2005)
9. Hirschkoff, D.: On the benefits of using the up-to techniques for bisimulation verification. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 285–299. Springer, Heidelberg (1999)
10. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Full semantics preservation in model transformation – a comparison of proof techniques. Technical Report TR-CTIT-10-09, CTIT, University of Twente (2010)
11. Kastenberg, H., Kleppe, A., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)
12. Königs, A.: Model transformation with triple graph grammars. In: Workshop on Model Transformations in Practice (2005)

13. Küster, J., Gschwind, T., Zimmermann, O.: Incremental development of model transformation chains using automated testing. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 733–747. Springer, Heidelberg (2009)
14. Leifer, J., Milner, R.: Deriving bisimulation congruences for reactive systems. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 243–258. Springer, Heidelberg (2000)
15. McComb, T., Smith, G.: Architectural Design in Object-Z. In: ASWEC 2004, pp. 77–86. IEEE, Los Alamitos (2004)
16. Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Trans. Software Eng. 30(2), 126–139 (2004)
17. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: GT-VMT 2006. ENTCS, vol. 211, pp. 185–194 (2006)
18. Necula, G.: Translation validation for an optimizing compiler. In: PLDI 2000. SIGPlan Notices, vol. 35, pp. 83–95. ACM, New York (2000)
19. Object Management Group: OMG Unified Modeling Language (OMG UML) – Superstructure, Version 2.2 (2009), `http://www.omg.org/docs/formal/09-02-02.pdf`
20. Pérez, J., Crespo, Y.: Exploring a method to detect behaviour-preserving evolution using graph transformation. In: Third International ERCIM Workshop on Software Evolution, pp. 114–122 (2007)
21. Rangel, G., König, B., Ehrig, H.: Deriving bisimulation congruences in the presence of negative application conditions. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 413–427. Springer, Heidelberg (2008)
22. Rangel, G., Lambers, L., König, B., Ehrig, H., Baldan, P.: Behavior preservation in model refactoring using DPO transformations with borrowed contexts. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 242–256. Springer, Heidelberg (2008)
23. Ruhroth, T., Wehrheim, H.: Refactoring object-oriented specifications with data and processes. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 236–251. Springer, Heidelberg (2007)
24. Schürr, A., Klar, F.: 15 years of triple graph grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
25. van Glabbeek, R.: The linear time - branching time spectrum II. In: Best, E. (ed.) CONCUR 1993. LNCS, vol. 715, pp. 66–81. Springer, Heidelberg (1993)
26. van Kempen, M., Chaudron, M., Kourie, D., Boake, A.: Towards proving preservation of behaviour of refactoring of UML models. In: SAICSIT 2005, pp. 252–259 (2005)