Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Conflict Detection for Model Versioning Based on Graph Modifications: Long Version

Gabriele Taentzer[1], Claudia Ermel[2],
Philip Langer[3] and Manuel Wimmer[4]

[1]Philipps-Universität Marburg, Germany
taentzer@mathematik.uni-marburg.de
[2]Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de
[3]Johannes-Kepler-Universität Linz, Austria
philip.langer@jku.at
[4]Technische Universität Wien, Austria
wimmer@big.tuwien.ac.at

Bericht-Nr. 2010-09
ISSN 1436-9915

Conflict Detection for Model Versioning Based on Graph Modifications: Long Version

Gabriele Taentzer[1], Claudia Ermel[2],
Philip Langer[3], and Manuel Wimmer[4]

[1] Philipps-Universität Marburg, Germany, taentzer@mathematik.uni-marburg.de
[2] Technische Universität Berlin, Germany, claudia.ermel@tu-berlin.de
[3] Johannes-Kepler-Universität Linz, Austria, philip.langer@jku.at
[4] Technische Universität Wien, Austria, wimmer@big.tuwien.ac.at

Abstract. In model-driven engineering, models are primary artifacts and can evolve heavily during their life cycle. Therefore, versioning of models is a key technique which has to be offered by an integrated development environment for model-driven engineering. In contrast to text-based versioning systems we present an approach which takes abstract syntax structures in model states and operational features into account. Considering the abstract syntax of models as graphs, we define model revisions as graph modifications which are not necessarily rule-based. Building up on the DPO approach to graph transformations, we define two different kinds of conflict detection: (1) the check for operation-based conflicts, and (2) the check for state-based conflicts on results of merged graph modifications.

1 Introduction

A key benefit of model-driven engineering is the management of the complexity of modern systems by abstracting its compelling details using models. Like source code, models may heavily evolve during their life cycle and, therefore, they have to be put under version control. Especially optimistic versioning is of particular importance because it allows for concurrent modifications of one and the same artifact performed by multiple modelers at the same time. When concurrent modifications are endorsed, contradicting and inconsistent changes, and therewith versioning conflicts, might occur. Traditional version control systems for code usually work on file-level and perform conflict detection by line-oriented text comparison. When applied to the textual serialization of models, the result is unsatisfactory because the information stemming from the graph-based structure is destroyed and associated syntactic and semantic information is lost.

To tackle this problem, dedicated model versioning systems have been proposed [1,2,3,4]. However, a uniform and effective approach for precise conflict detection and supportive conflict resolution in model versioning still remains

an open problem. For the successful establishment of dedicated model versioning systems, a profound understanding by means of formal definitions of potentially occurring kinds of conflicts is indispensable, but yet missing.

Therefore, we present a formalization of two different kinds of conflicts based on graph modifications. We introduce this new notion of graph modifications to generalize graph transformations. Graph modifications are not necessarily rule-based, but just describe changes in graphs. Two kinds of conflict detection are defined based on graph modifications: (1) operation-based conflicts and (2) state-based conflicts. The specification of operations is based on rules. Therefore, we extract minimal rules from graph modifications and/or select suitable pre-defined operations and construct graph transformations in that way. Conflict detection is then based on parallel dependence of graph transformations and the extraction of critical pairs as presented in [5]. State-based conflicts are concerned with the well-formedness of the result after merging graph modifications. To detect state-based conflicts, two graph modifications are merged and the result is checked against pre-defined constraints. The proposed critical pair extraction is not complex and corresponds to the procedure of textual versioning systems, i.e. they are applied whenever graph modifications have occurred and are being checked in. For each check-in, all those modifications (check-ins) are checked for conflicts that have taken place since the check-out of the model performed by the user who is performing the current check-in.

The paper is structured as follows: In Section 2, we introduce the concept of graph modification and recall the notion of graph transformation. While Section 3 is concerned with the detection of operation-based conflicts, Section 4 presents the detection of state-based conflicts. Sections 5 and 6 discuss implementation issues and related work and finally, Section 7 concludes this paper.

2 Graph Modifications and Graph Transformations

Throughout this paper, we describe the underlying structure of a model by a graph. To capture all important information, we use typed, attributed graphs and graph morphisms as presented in [5]. In the following, we omit the terms "typed" and "attributed" when mentioning graphs and graph morphisms.

All model modifications are considered on the level of the abstract syntax where we deal with graphs. We introduce graph modifications generalizing the concept of graph transformation to graph changes which are not necessarily

rule-based. A graph modification is a partial injective graph mapping being defined by a span of injective graph morphisms.

Definition 1 (Graph modification). *Given two graphs G and H, a* direct graph modification *$G \Longrightarrow H$ is a span of injective morphisms $G \xleftarrow{g} D \xrightarrow{h} H$. A sequence $G = G_0 \Longrightarrow G_1 \Longrightarrow \ldots \Longrightarrow G_n = H$ of direct graph modifications is called* graph modification *and is denoted by $G \overset{*}{\Longrightarrow} H$.*

Graph D characterizes an intermediate graph where all deletion actions have been performed but nothing has been added yet.

Example 1 (Graph and graph revision). Consider the following model versioning scenario for statecharts. The abstract syntax of a statechart can be defined by a typed, attributed graph, as e.g. shown in Fig. 1 (a). The node type is given in the top compartment of a node. The name of each node of type State is written in the attribute compartment below the type name. We model hierarchical statecharts by using containment edges. For instance, in Fig. 1 (a), state S0 contains S1 and S2 as substates. (In contrast to UML state machines, we distinguish these edges which present containment links by composition decorators.) Note that we abstract from transition events, guards and actions, as well as from other statechart features. Furthermore, from now on we use a compact notation of the abstract syntax of statecharts, where we draw states as nodes (rounded rectangles with their names inside) and transitions as directed arcs between state nodes. The compact notation of the statechart in Fig. 1 (a) is shown in Fig. 1 (b). Fig. 1 (c) shows the statechart in the well-known concrete syntax.

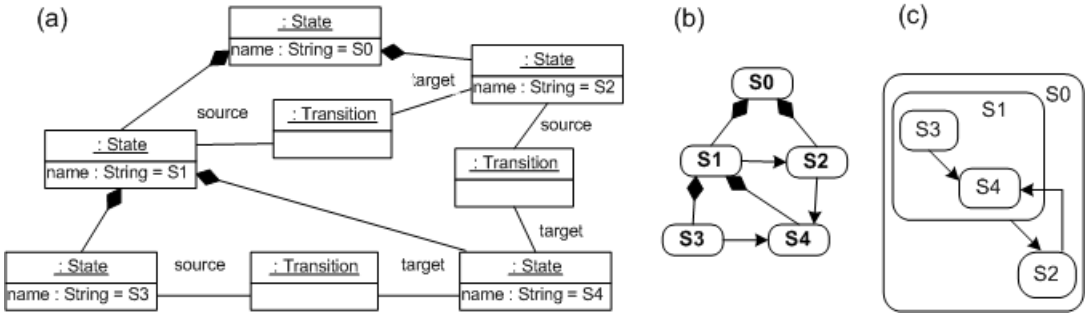


Fig. 1. Sample statechart: abstract syntax graph (a), compact notation (b) and concrete syntax (c)

In our scenario for model versioning, three users check out the current statechart shown in Fig. 1 and change it in three different ways. User A intends

to perform a refactoring operation on it. She moves state S3 up in the state hierarchy (cf. Fig. 2).

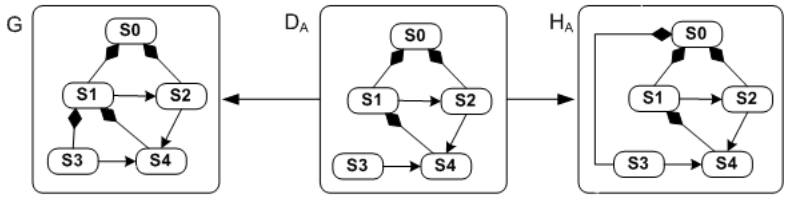


Fig. 2. Refactoring step as graph modification gm_A

User B refines the statechart by adding a new state S5 inside superstate S0 and connects this newly added state S5 to state S2 in the same superstate by drawing a new transition between them. Moreover, the transition connecting S2 to S4 is deleted in this refinement step. This graph modification is shown in Fig. 3. Finally, user C deletes state S3 together with its adjacent transition to state S4 (cf. Fig. 4).

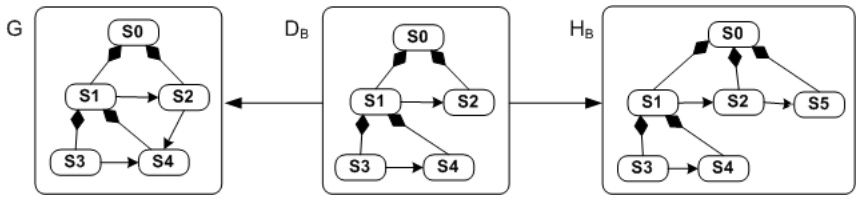


Fig. 3. Refinement step as graph modification gm_B

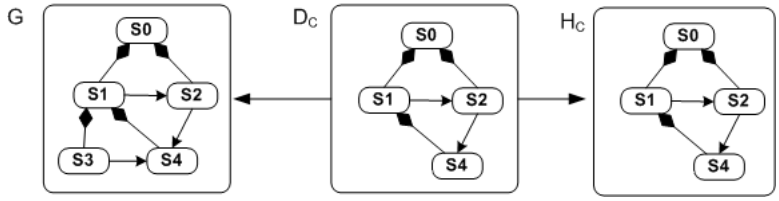


Fig. 4. Deletion step as graph modification gm_C

Obviously, conflicts occur when these users try to check in their changes: state S3 is deleted by user C but is moved to another container by user A.

Furthermore, user B and user C delete different transitions adjacent to state S4. This may lead to a problem if the statechart language forbids isolated states (not adjacent to any transition), although each single change does not create a forbidden situation.

Since we will use rules to detect conflicts between graph modifications, we recall the notions of graph rule and transformation here. We use the DPO approach for our work, since its comprehensive theory as presented in [5] is especially useful to formalize conflict detection in model versioning.

Definition 2 (Graph rule and transformation). *A* graph rule $p = L \xleftarrow{l} K \xrightarrow{r} R$ *consists of graphs* L, K *and* R *and injective graph morphisms* l *and* r*. Given a match* $m\colon L \to G$*, graph rule* p *can be applied to* G *if a double-pushout (DPO) exists as shown in the diagram below with pushouts* (PO_1) *and* (PO_2) *in the category of typed, attributed graphs. Graph* D *is the intermediate graph after removing* $m(L)$*, and* H *is constructed as gluing of* D *and* R *along* K *(see [5]).* $G \overset{p,m}{\Longrightarrow} H$ *is called* graph transformation.

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO_1) & \downarrow{\scriptstyle k} & (PO_2) & \downarrow{\scriptstyle m'} \\ G & \xleftarrow{g} & D & \xrightarrow{h} & H \end{array}$$

Obviously, each graph transformation can be considered as graph modification by forgetting about the rule and its match. If the rule and its match are given, the pushout (PO_1) has to be constructed as pushout complement. We recall the definition of a pushout complement as presented in [5]. From an operational point of view, a pushout complement determines the part in graph G that does not come from L, but includes K.

Definition 3 (PO complement). *Given morphisms* $l\colon K \to L$ *and* $m\colon L \to G$*, then* $k\colon K \to D$ *and* $g\colon D \to G$ *is the pushout complement (POC) of* l *and* m*, if* (PO_1) *in Def. 2 is a pushout.*

3 Detection of Operation-Based Conflicts

For the detection of operation-based conflicts, we have to find out the operation resulting in a particular graph modification. Thus, we have to find a corresponding rule and match, i.e. a corresponding graph transformation to the given graph modification. An approach which is always possible is to extract a minimal rule [6], i.e. a minimal operation which contains all atomic actions (i.e. creation and deletion of nodes, edges, and attribute values) that are performed by the given graph modification. Thus, the extraction of a minimal rule together with its match leads to a minimal graph transformation performing a given graph modification.

If the operation which led to a graph modification is not known but can be specified by a graph rule, a suitable method to identify the right operation is to extract again the minimal rule and to find the corresponding operation (out of a set of pre-defined operations) by comparing it with the minimal rule.

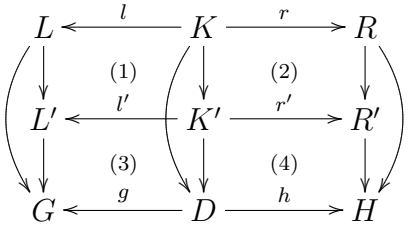
After having specified graph modifications by graph transformations, the parallel independence of transformations can be checked and critical situations are identified as conflicts. These conflicts can be specified as critical pairs [5].

3.1 Extraction of a minimal rule

As first step, we use the construction of minimal rules by Bisztray et al. [6]. This construction yields in a natural way a minimal DPO rule for a graph modification. Minimal rules contain the proper atomic actions on graphs with minimal contexts. Bisztray et al. have shown that this construction is unique, i.e. no smaller rule than the minimal rule can be constructed for a given graph modification.

Definition 4 (Minimal graph rule and transformation). *Rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R$ *is* minimal *over direct graph modification* $G \xleftarrow{g} D \xrightarrow{h} H$ *if the outer DPO exists and for each*

rule $L' \xleftarrow{l'} K' \xrightarrow{r'} R'$ *with injective morphism* $K' \to D$ *and pushouts (3) and (4), there are unique morphisms* $L \to L'$, $K \to K'$, *and* $R \to R'$ *such that the following diagram commutes and (1), (2), (1) + (3), and (2) + (4) are pushouts. Graph transformation* $G \overset{p}{\Longrightarrow} H$ *is also called* minimal.



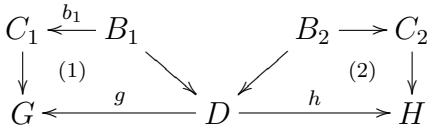
The following minimal rule construction extracts all deletion and creation actions from a given transformation in graphs L_1 and R_1 by constructing so-called initial pushouts. In a nutshell, an initial pushout extracts a graph morphism consisting of the changing part of the given graph morphism, i.e. the non-injective mapping part as well as the codomain part that is not in the image of the morphism (see [5]). This is done for both sides of a graph modification, leading to the left and the right-hand sides of the minimal rule. In the middle, two gluing graphs are constructed which have to be glued together, and the left and the right-hand sides are potentially extended by further necessary context.

Definition 5 (Initial pushout). *Let $g\colon D \to G$ be a graph morphism, an* initial pushout *over g consists of graph morphisms $c_1\colon C_1 \to G$, $b_1\colon B_1 \to C_1$, and $d_1\colon B_1 \to D$ (cf. diagram below) such that g and c_1 are a pushout over b_1 and d_1. For every other pushout over g consisting of $c_1'\colon C_1' \to G$, $b_1'\colon B_1' \to C_1'$, and injective $d_1'\colon B_1' \to D$, there are unique graph morphisms $b\colon B_1 \to B_1'$ and $c\colon C_1 \to C_1'$ such that $c_1' \circ c = c_1$ and $b_1' \circ b = b_1$. Moreover, (c, b_1') is a pushout over (b_1, b).*

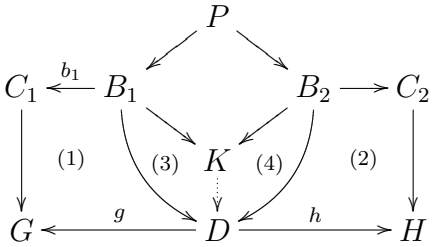
Note that for graph morphisms, initial pushouts do always exist [5].

Definition 6 (Construction of Minimal Rules from Graph Modifications). *Given a graph modification $G \leftarrow D \rightarrow H$, the* minimal rule *$L \leftarrow K \rightarrow R$ is constructed as follows:*

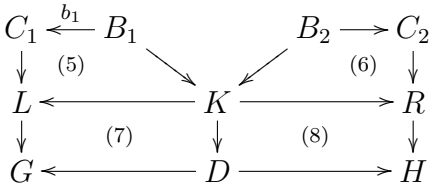
1. *Construct the initial pushouts (1) of $D \to G$ and (2) of $D \to H$*



2. *Construct the pullback $B_1 \leftarrow P \rightarrow B_2$ of $B_1 \rightarrow D \leftarrow B_2$, and then the pushout $B_1 \rightarrow K \leftarrow B_2$ of $B_1 \leftarrow P \rightarrow B_2$, leading to unique injective morphism $K \rightarrow D$ such that $(3), (4)$ commute.*



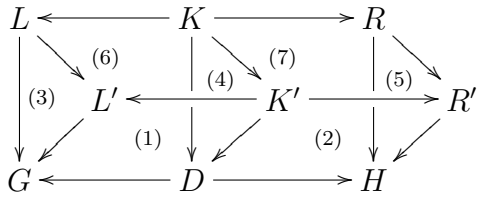
3. *Finally split initial pushouts (1) and (2) by (3) and (4), i.e. construct pushouts (5) of $C_1 \leftarrow B_1 \rightarrow K$ and (6) of $K \leftarrow B_2 \rightarrow C_2$, leading to pushouts (7) and (8) by pushout decomposition, and the minimal rule $L \leftarrow K \rightarrow R$.*



The DPO $(7), (8)$ is called minimal transformation of $G \leftarrow D \rightarrow H$ via the minimal rule $L \leftarrow K \rightarrow R$.

Proposition 1 (Minimal rule). *Given any rule $L' \leftarrow K' \rightarrow R'$ which generates $G \leftarrow D \rightarrow H$ with injective match via pushouts (1) and (2) below, and*

the outer diagram is a DPO with minimal rule and minimal transformation. Then, there are unique injective morphisms $L \to L'$, $K \to K'$ and $R \to R'$ such that $(3) - (5)$ commute and $(6), (7)$ are pushouts.



Proof. The proof of Prop. 1 is given in [6]. Note that after Step 1, the minimal rule extraction may also be considered as E-concurrent rule constructed from a deletion rule on the left and a creation rule on the right.

Example 2 (Minimal rule construction). The construction of the minimal rule p_A for graph modification gm_A (the refactoring performed by user A in Fig. 2) is depicted in Fig. 5. Note that the minimal rule does not contain any attributes, since they are not changed within the graph modification. The minimal rules

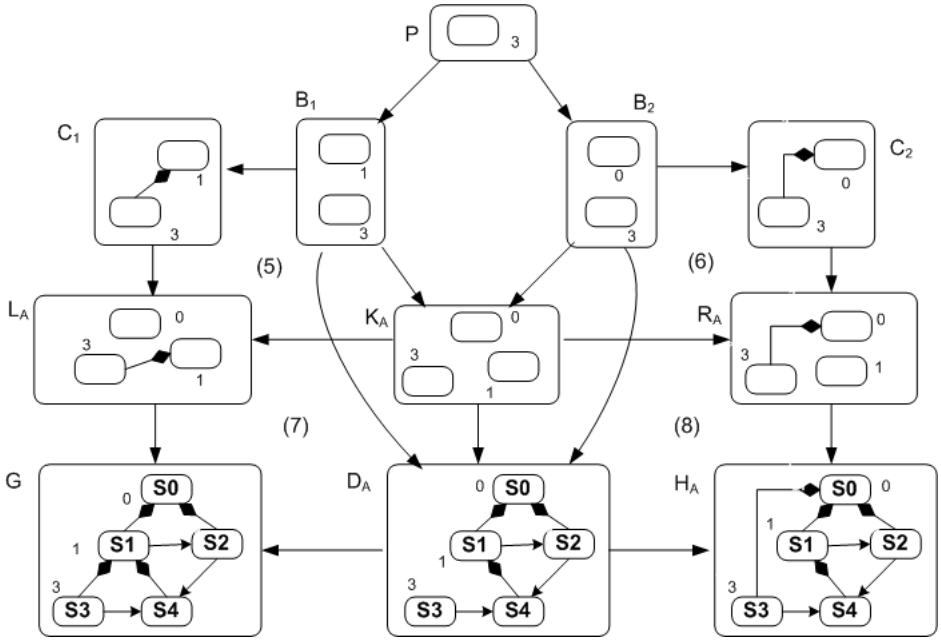


Fig. 5. Construction of minimal rule $p_A = (L_A \leftarrow K_A \rightarrow R_A)$ for gm_A

p_B for gm_B (the refinement performed by user B in Fig. 3) and p_C for gm_C (the deletion by user C in Fig. 4) are constructed analogously. The results are

depicted in Figs. 6 and 7. Note that the initial pushout construction leads to variables as attribute values of deleted and newly created nodes.

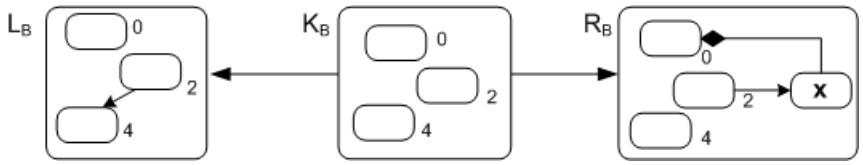


Fig. 6. Minimal rule p_B for graph modification gm_B

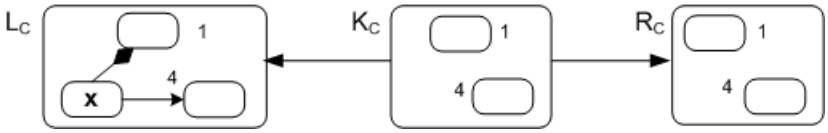


Fig. 7. Minimal rule p_C for graph modification gm_C

Comparing the applications of minimal rules p_A and p_C, we see that minimal rule p_C deletes state `S3` that is used by minimal rule p_A to perform its refactoring. Such conflicts are called Delete/Use-conflicts and are defined below. They can be automatically detected using the graph transformation tool AGG [7].

3.2 Identification of operations

Up to now, we investigated the actual actions performed by different users which we extracted in minimal rules. This approach does not take any pre-defined operations into account. Given a set of change operations defined by graph rules, we can identify the right operation that has been performed for a graph modification gm by the following method: we extract again the minimal rule for gm and find the corresponding operation (out of a set of pre-defined operations) by comparing with the minimal rule. An operation o is *executable* wrt. a given graph modification $G \longleftarrow D \longrightarrow H$ if the extracted minimal rule is a subrule of operation o and o is applicable to the original graph G in a compatible way.

Definition 7 (Subrule). *A rule $L_s \xleftarrow{l_s} K_s \xrightarrow{r_s} R_s$ is a subrule of rule $L \xleftarrow{l} K \xrightarrow{r} R$ if morphisms $i_l\colon L_s \to L$, $i_k\colon K_s \to K$, and $i_r\colon R_s \to R$ exist and the diagram on the right commutes and (PO_1) and (PO_2) are pushouts.*

$$\begin{array}{ccccc} L_s & \xleftarrow{l_s} & K_s & \xrightarrow{r_s} & R_s \\ {\scriptstyle i_l}\downarrow & (PO_1) & {\scriptstyle i_k}\downarrow & (PO_2) & \downarrow{\scriptstyle i_r} \\ L & \xleftarrow{l} & K & \xrightarrow{r} & R \end{array}$$

Definition 8 (Minimal rule-related operation execution). *Given a minimal rule $s = L_s \xleftarrow{l_s} K_s \xrightarrow{r_s} R_s$ applicable to graph G by match $m_s : L_s \to G$, an operation given by rule $o = L \xleftarrow{l} K \xrightarrow{r} R$ is executable on graph G wrt. rule s if s is a subrule of o (as defined in Def. 7) and if o is applicable to G by a match $m\colon L \to G$ such that $m \circ i_l = m_s$ with $i_l\colon L_s \to L$.*

Note that Def. 8 is useful for identifying single operations per minimal rule. It cannot be used to identify sequences of operations which relate to one minimal rule. This problem is left to future work.

Example 3. We define an operation enabling the user to move a state s to another container only if the new container state contains the previous container state of s. This avoids producing cyclic containments. The operation rule o_A used for graph modification gm_A has more context than its minimal rule (cf. Fig. 8 for the relation between these two rules where the minimal rule p_A is shown in the upper half and the operation rule o_A in the lower half).

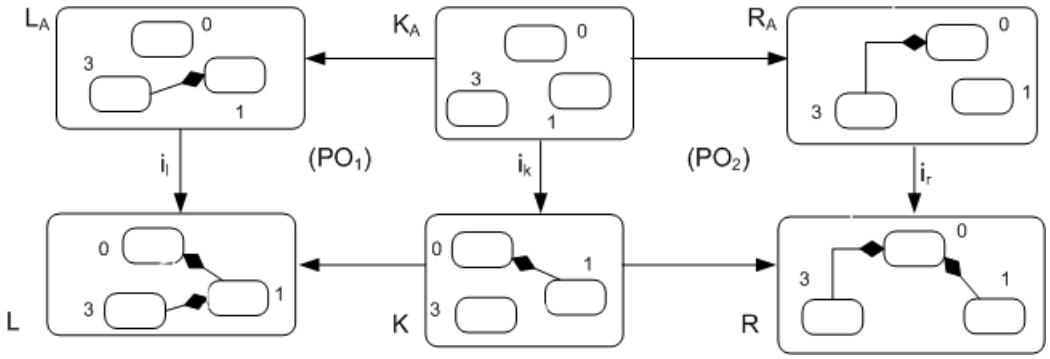


Fig. 8. *MoveState* operation

Once the executed operations have been identified, we can start the conflict detection also for these rules. Since they can come with more context, more conflicts might occur compared to the conflict detection based on minimal rules.

3.3 Detection of operation-based conflicts

After having constructed graph transformations from graph modifications by minimal rule extraction and/or operation rule selection, we can use critical pairs as presented in [5] to define operation-based conflicts.

First we check if two graph transformations are parallel independent which means that rule matches overlap in preserved items only.

Definition 9 (Parallel independent transformations). *Two direct graph transformations* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *being applications of rules* $p_1 = (L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_1} R_1)$ *and* $p_2 = (L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2)$ *at matches* $m_1 : L_1 \to G$ *and* $m_2 : L_2 \to G$ *are* parallel independent *if the transformations preserve all items in the intersection of both matches, i.e.* $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$.

To concentrate on the proper conflict, we abstract from unnecessary context of identified parallel dependent transformations. This leads to the notion of a *critical pair* which is defined below. A critical pair consists of two parallel dependent transformations starting from a smaller graph K now. K can be considered as a suitable gluing of left-hand sides L_1 and L_2 of corresponding rules. For each two parallel dependent transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ a corresponding critical pair can be found. We consider critical pairs as operation-based conflicts.

Definition 10 (Critical pair). *A* critical pair *consists of two parallel dependent graph transformations* $K \overset{p_1,o_1}{\Longrightarrow} P_1$ *and* $K \overset{p_2,o_2}{\Longrightarrow} P_2$ *with matches* $o_1 \colon L_1 \to K$ *and* $o_2 \colon L_2 \to K$ *being jointly surjective.*

Proposition 2 (Completeness of critical pairs). *Given two direct graph transformations* $G \overset{p_1,m_1}{\Longrightarrow} H_1$ *and* $G \overset{p_2,m_2}{\Longrightarrow} H_2$ *which are parallel dependent, there is a* critical pair $K \overset{p_1,o_1}{\Longrightarrow} P_1$ *and* $K \overset{p_2,o_2}{\Longrightarrow} P_2$ *such that there is a graph morphism* $o \colon K \to G$ *with* $o \circ o_1 = m_1$ *and* $o \circ o_2 = m_2$.

The proof can be found in [5]. Note that this proposition is very useful to find operation-based conflicts. If the pair of transformations considered is parallel dependent, we get a critical pair, i.e. an operation-based conflict. By proposition 2 we know that we get all operation-based conflicts that way.
This formalization enables us to detect so-called *Delete/Use conflicts:* one transformation deletes a graph item while the other one reads it. Note that a particular kind of delete/use conflicts are sometimes called *Change/Use conflicts*. Here, the first transformation changes the value of an attribute, while the

second one either changes it too, or just checks its value. Since attribute value bindings are modeled by edges, attribute value changes involve the deletion of edges (cf. [5]).

In case we find critical pairs, we can identify the conflict in form of the minimal context of the critical match overlappings. Note that since we have given the overlapping of the left-hand sides of the minimal rules already, we need to check only this overlapping situation for a conflict. This procedure needs much less effort than the normal critical pair analysis in AGG which computes all possible contexts.

Example 4. Considering the minimal rules p_A and p_C in Figs. 5 and 7 applied to graph G in Figs. 2 - 4 with the obvious matches, we get a Delete/Use-conflict based on deletion and usage of state S3.

4 Detection of state-based conflicts

Besides *operation-based* conflicts, we want to detect *state-based* conflicts which can occur in merged modification results. These conflicts occur if a merged modification result shows some abnormality not present in the modification results before merging. Detection of state-based conflicts is done by constraint checking. The constraints may be language-specific, i.e. potentially induced by the corresponding graph language definition.

In the following, we present a procedure to merge two different graph modifications to find state-based conflicts. We show that this merge procedure yields the expected result if there are no operation-based conflicts. Otherwise, the procedure cannot be performed. Thus, a natural ordering of conflict detection is

1. to extract minimal rules and analyze minimal transformations for operation-based conflicts,
2. to find further operation-based conflicts by analyzing operations,
3. to check for state-based conflicts after all operation-based conflicts have been resolved.

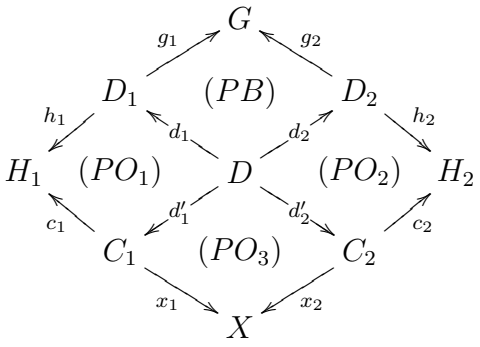
4.1 Merging of graph modifications

To determine the merge graph of two graph modifications, the "least common denominator" of both modifications is determined. It is called D in the construction below. Considering both modifications, the least common denominator is extended by all the changes of modifications 1 and 2, first separately

in C_1 and C_2, and finally glued together in graph X. (Compare the diagram in Def. 11.)

Definition 11 (Merging of graph modifications). *Given two modifications $G \xleftarrow{g_1} D_1 \xrightarrow{h_1} H_1$ and $G \xleftarrow{g_2} D_2 \xrightarrow{h_2} H_2$, the merged graph X and the merged graph modification $G \longleftarrow D \longrightarrow X$ can be constructed as follows:*

1. *Construct $D_1 \xleftarrow{d_1} D \xrightarrow{d_2} D_2$ as pullback of $D_1 \xrightarrow{g_1} G \xleftarrow{g_2} D_2$.*
2. *Construct PO-complements $d'_i \colon D \to C_i$ and $c_i \colon C_i \to H_i$ from morphisms $d_i \colon D \to D_i$ and $h_i \colon D_i \to H_i$ for $i = 1, 2$.*
3. *Construct $C_1 \xrightarrow{x_1} X \xleftarrow{x_2} C_2$ as pushout of $C_1 \xleftarrow{d'_1} D \xrightarrow{d'_2} C_2$.*
4. *The merged graph modification is $G \xleftarrow{g_1 \circ d_1} D \xrightarrow{x_1 \circ d'_1} X$.*



Proposition 3 (Existence and uniqueness of merging). *The construction in Def. 11 leads to a unique graph X and furthermore, unique graph modification $G \xleftarrow{g_1 \circ d_1} D \xrightarrow{x_1 \circ d'_1} X$ up to isomorphism, if graph transformations $G \overset{p_1, m_1}{\Longrightarrow} H_1$ and $G \overset{p_2, m_2}{\Longrightarrow} H_2$ with minimal rules $p_1 = L_1 \xleftarrow{l_1} K_1 \xrightarrow{r_2} R_1$ and $p_2 = L_2 \xleftarrow{l_2} K_2 \xrightarrow{r_2} R_2$ uniquely extracted from graph modifications $G \xleftarrow{g_1} D_1 \xrightarrow{h_1} H_1$ and $G \xleftarrow{g_2} D_2 \xrightarrow{h_2} H_2$ are parallel independent.*

Proof. 1. *Existence:* While pullback and pushouts always exist in the category of graph and graph morphisms, this is not the case for pushout complements. We show the existence of pushout complements in PO_1 and PO_2 by contraposition: If the complements POC_1 of PO_1 or POC_2 of PO_2 do not exist, there is not a morphism $L_1 \to D_2$ or a morphism $L_2 \to D_1$.
If POC_1 does not exist, the gluing condition is not satisfied for d_1 and h_1 (compare Fact 3.11 in [5]). Since h_1 and d_1 are injective, this means that the dangling condition is not satisfied. Thus, there is an edge $e \in H_{1E}$ without origin in D_1 but with an adjacent node $h_1(x)$ with $x \in D_{1N}$ and x has not an origin in D. Of course, there is $g_1(x) = y \in G$. Since D is a PB-object, there does not exist a node in D_2 being mapped to y by g_2.
Since there is an edge $e \in H_1$ without origin in D_1, it must have an origin in R_1 without origin in K_1, since a minimal graph transformation is a double pushout. Moreover, $x \in D_1$ has an origin in K_1 which has an image $l \in L_1$ mapped to $y \in G$. However, $l \in L_1$ cannot be mapped to some node in D_2

such that it would be mapped to $y \in G$. Thus, there is not a morphism $n_1 \colon L_1 \to D_2$ such that $g_2 \circ n_1 = m_1$.
Analogously, we can show that there is not a morphism $L_2 \to D_1$, if POC_2 does not exist.

2. *Uniqueness:* Pullback and pushout constructions are unique up to isomorphism. Moreover, POC constructions are unique up to isomorphism, if the gluing condition is satisfied (compare again Fact 3.11 in [5]).

Example 5 (Merging of graph modifications). The merging construction applied to gm_B and gm_C (the refinement and deletion modifications by users B and C) is depicted in Fig. 9. We see that the merged graph X at the bottom of Fig. 9 contains a forbidden situation: state S4 is isolated, i.e. it is not adjacent to a transition anymore. We want to find such a situation automatically by checking a graph constraint forbidding any situation where a state is isolated.

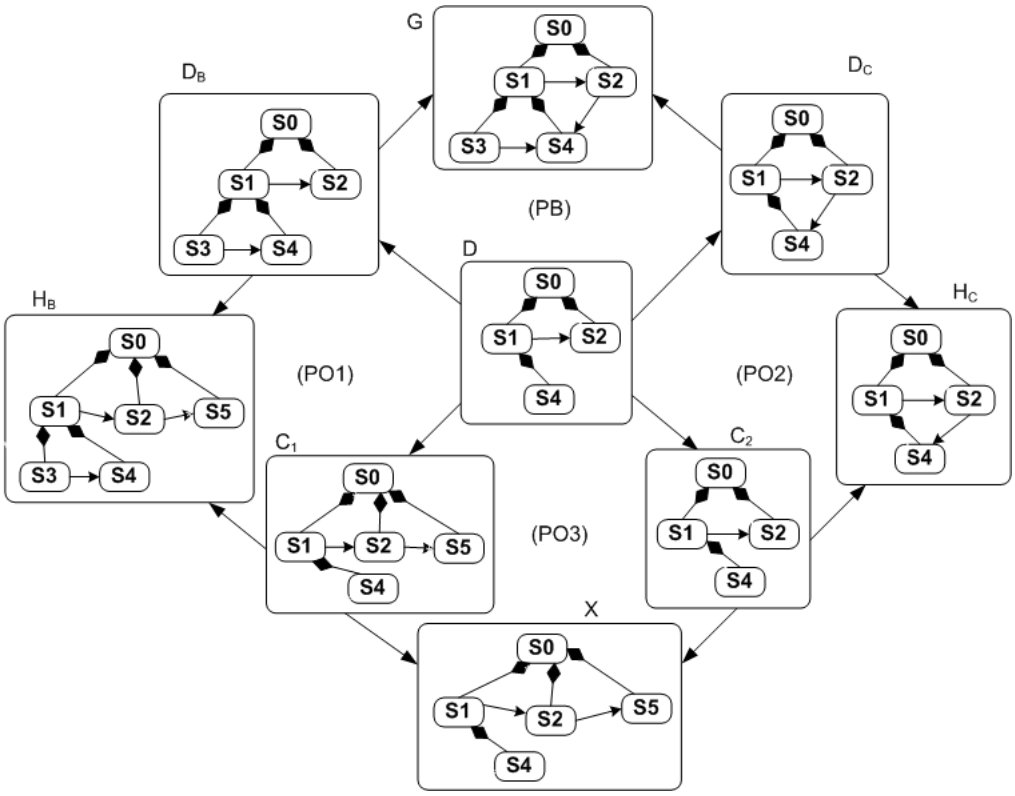


Fig. 9. Merging of graph modifications gm_B and gm_C

4.2 Detection of state-based conflicts

Using graph conditions as defined in [8], we can specify well-formedness constraints to be checked on the merged graph.

Definition 12 (Graph condition and graph constraint). *A* graph condition *over graph G is of the form* true *or $\exists(a, c)$ where $a : P \to C$ is a graph morphism and c is a condition over C. Moreover, Boolean formulas over conditions over P yield conditions over P, i.e. $\neg c$ and $\wedge_{j \in J} c_j$ are (Boolean) conditions over P where J is a finite index set and c, $(c_j)_{j \in J}$ are conditions over P. Additionally, $\exists a$ abbreviates $\exists(a, true)$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, $false$ abbreviates $\neg true$, $\vee_{j \in J} c_j$ abbreviates $\neg \wedge_{j \in J} \neg c_j$, and $c \implies d$ abbreviates $\neg c \vee d$.*

Every graph morphism satisfies *true. A morphism $p : P \to G$ satisfies condition $\exists(a, c)$ if there is an injective graph morphism $q : C \to G$ such that $q \circ a = p$ and q satisfies c. A graph G satisfies a condition $\exists(a, c)$ if this condition is satisfied by graph morphism $\emptyset \to G$. In the context of graphs, graph conditions are called* graph constraints*. The satisfaction of conditions by graphs and morphisms is extended to Boolean conditions in the usual way.*

The notation of graph constraints of the form $\exists(a : \emptyset \to G, c)$ can be shortened to $\exists(G, c)$ without loss of information. A rule application condition of the form $\neg\exists a$ is usually called *negative application condition* (see [8]).

Definition 13 (State-based conflict). *Given a merged graph modification as in Def. 11, a* state-based conflict *$(C, H_1 \Longrightarrow X, H_2 \Longrightarrow X)$ consists of a graph constraint C, and graph modifications $H_1 \Longrightarrow X$ and $H_2 \Longrightarrow X$ such that C is satisfied by graphs H_1 and H_2 but not by X.*

Example 6. Fig. 9 shows a merged graph X that contains an isolated state which should not be allowed. This situation can be formalized by a graph constraint $C = \forall G_0((\exists a : G_0 \to G_1) \vee (\exists a : G_0 \to G_2))$ where G_0 consists of a state contained in some other state, and G_1 and G_2 show the alternative required contexts for G_0 in Fig. 10. C is satisfied by graphs H_B and H_C in Fig. 9, but not by graph X.

Further typical state-based conflicts can occur w.r.t. well-formedness constraints in meta models. Consider e.g. the constraint that a transition may have at most one event. If two graph modifications add an event to the same transition, then this leads to a state-based conflict after merging.

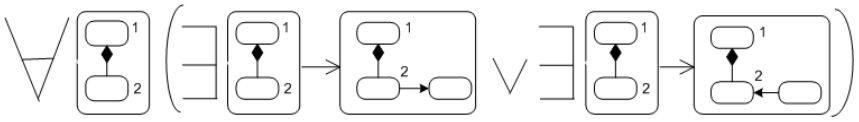


Fig. 10. Graph constraint forbidding isolated states

5 Implementation issues

5.1 AGG

AGG [5] is an integrated development environment for algebraic graph transformation systems. It implements basic concepts and constructions such as graphs, graph morphisms, matching, and pushouts. On top of these, graph transformation, a restricted form of graph constraint checking as well as parallel dependency checking of rule applications are supported. This is a solid basis to implement initial pushouts, pullbacks, and pushout complements as basis for minimal rule extraction, merging and conflict detection with a convenient user interface in the near future.

5.2 AMOR.

We now compare the model versioning system AMOR [1] with the formal definitions presented in this paper.

Model differencing. Before conflicts are detected, the concurrently performed changes have to be determined. In this paper, the modifications are extracted applying the construction of minimal rules (cf. Def. **??**). To identify executions of predefined operations, it is checked whether the extracted minimal rule is a subrule of a specific predefined operation. In AMOR, all applied changes are derived by conducting EMF Compare [9]. The resulting difference model is conceptually equivalent to a minimal graph rule, because it consists of a match model explicating common elements (cf. D in a graph modification) and a set of atomic changes describing the differences between G and H. Executions of predefined operations are detected in AMOR by searching for their specific change patterns in the derived difference model and, subsequently for each match, by evaluating their pre- and postconditions. Again, this technique is compliant to the operation identification presented in this paper.

Operation-based conflicts. As proposed in Sect. 3, we check minimal rules and operations for critical pairs which are considered as operation-based conflicts. In AMOR, operation-based conflicts are detected by comparing each change applied by one user to each change applied by another. If two changes

are contradicting, a conflict explicating the contradicting changes is reported. If further operation executions have been identified before, the preconditions of this operation are evaluated after all atomic changes of the opposite side are executed. With this it is checked whether the operation may still be executed (according to its preconditions) to a model incorporating the opposite changes.

State-based conflicts. These conflicts occur if a merged model bears prohibited conditions. In this paper, such conditions are defined by graph constraints which are evaluated on the merged graph (cf. Sec. 4.1). According to Proposition 3, this is possible only, if there are no parallel dependent modifications. In AMOR, the merged model is validated against the metamodel and its OCL constraints to find state-based conflicts as well. However, the applied merge differs slightly from the merge presented in this paper, since a merged model is also created, if parallel dependent changes exist. In such cases, only those changes are propagated to the merged model which are not parallel dependent and the user has to manually resolve the operation-based conflicts.

6 Related Work

The main contribution of this work is a formal definition of model versioning conflicts as basis for automatic conflict detection by using the DPO approach to graph transformations. Therefore, we distinguish two kinds of related work. First, we discuss the state-of-the-art of current model versioning systems, and second, we compare our work to other approaches aiming at the formalization of model versioning conflicts.

Model Versioning Systems. In the last decades a lot of research has been conducted in the domain of software versioning which is profoundly outlined in [10,11]. Most of the approaches focus on source code versioning, others focus on two-way comparison of models [12], but there are also some dedicated approaches aiming at the versioning of models by a three-way merge. For example, Odyssey-VCS [2] supports the versioning of UML models. This system performs the conflict detection at a very fine-grained level, hence it is able to merge modifications concerning different model elements or even different attributes of one model element. EMF Compare [9] is an Eclipse plug-in, for comparing and merging models independently of the underlying meta model. CoObRA [4] is integrated in the Fujaba tool suite and logs the changes performed on a model. The modifications performed by the modeler who did the later commit are replayed on the updated version of the repository. Conflicts are reported if an operation may not be applied due to a violated precondition.

Similar to CoObRA, Unicase [3] also provides three-way merging based on edit logs.

Although *delete/use conflicts* and *change/use conflicts* are captured by all of these systems, they do not take predefined operations like refactorings and, consequently, their bigger contexts into account. EMF Compare and Unicase also miss to detect changes causing state-based conflicts. None of the four mentioned systems aims at providing a precise formalization of conflict detection.

Formalization of versioning conflicts. Another category-theoretical approach which formalizes model versioning is given in [13]. Similarly to our approach, modifications are considered as spans of morphisms to describe a partial mapping of models and model merging is defined by pushout constructions. Moreover, syntactic conflicts such as adding structure to an element which has been deleted, are identified. This kind of conflicts is very close to our delete/use-conflicts which we can identify after having extracted minimal rules. In contrast to [13], we refer to a formal analysis of operation-based conflicts. In addition, we consider state-based conflict detection. This has been indicated as future work in [13], where conflict detection based on user-specified operations are not mentioned at al.

Alanen and Porres [14] define a difference and merge operator for MOF-based models from a set-theoretical view. Differences are represented by atomic changes leading from a base version to the working copy. With their approach, they are able to detect *Delete/Use* and *Change/Use-conflicts*, also incorporating advanced concepts such as ordered features. However, conflicts going beyond atomic changes as well as state-based conflicts remain undetected.

The approach by Blanc et al. [15,16] detects state-based inconsistencies using Prolog empowered first-order logics. Structural and methodological constraints are formalized in consistency rules as logic formulae over a sequence of model construction operations. However, they do not consider to detect operations going beyond atomic changes as it is supported by our approach.

7 Conclusion

Two different kinds of conflicts in model versioning are defined in this paper based on the notion of graph modifications: *operation-based* and *state-based* conflicts. Graph modifications are not necessarily rule-based, but just describe changes in graphs. Operation-based conflicts are detected by extracting minimal rules from modifications first and selecting pre-defined operation rules thereafter if possible. As a consequence, we can use the well-known conflict characterization for graph transformations based on parallel dependence

checking and extraction of critical pairs. The detection of state-based conflicts builds directly on merged graph modifications and constraint checking.

In this paper, operations are specified simply by rules without additional application conditions. Several extensions are imaginable here: If operations are specified by rules with negative application conditions, an additional kind of conflict can be identified namely *Produce/Forbid-conflicts*. New parallel independence results for rules with more complex application conditions and their applications are currently elaborated by Habel et al. Moreover, the detection of operation sequences for minimal rules is left to future work.

Throughout this paper, we concentrate on the formalization of *conflict detection*. What can *conflict resolution* mean in this setting? The resolution of an operation-based conflict means to show the confluence of the corresponding critical pair (see [5]), while state-based conflicts might be solved by the definition of repair actions. Usually, different conflict resolutions are possible and it is up to future work to develop adequate resolution strategies for this formal setting.

References

1. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kargl, H.: Adaptable Model Versioning in Action. In: Modellierung 2010. Volume 161 of LNI., GI (2010)
2. Murta, L., Corrêa, C., Prudêncio, J.G., Werner, C.: Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In: 2nd Int. Workshop on Comparison and Versioning of Software Models @ ICSE'08. (2008)
3. Kögel, M.: Towards Software Configuration Management for Unified Models. In: Workshop on Comparison and Versioning of Software Models @ ICSE'08. (2008)
4. Schneider, C., Zündorf, A., Niere, J.: CoObRA - A Small Step for Development Tools to Collaborative Environments. In: Workshop on Directions in Software Engineering Environments @ ICSE'04. (2004)
5. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer (2006)
6. Bisztray, D., Heckel, R., Ehrig, H.: Verification of architectural refactorings: Rule extraction and tool support. ECEASST **16** (2008)
7. TFS-Group, TU Berlin: AGG. (2009) http://tfs.cs.tu-berlin.de/agg.
8. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. in Comp. Sci. **19**(2) (2009) 245–296
9. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional (2008)
10. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys **30**(2) (1998) 232–282
11. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering **28**(5) (2002) 449–462
12. Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. In: Software Engineering 2005. Volume 64 of LNI., GI (2005) 105–116

13. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A Category-Theoretical Approach to the Formalisation of Version Control in MDE. In: Fundamental Approaches to Software Engineering (FASE'09). Volume 5503 of LNCS., Springer (2009) 64–78
14. Alanen, M., Porres, I.: Difference and union of models. In: UML 2003—The Unififed Modeling Language. Volume 2863 of LNCS., Springer (2003) 2–17
15. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: ICSE'08—30th Int. Conference on Software Engineering. (2008)
16. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental Detection of Model Inconsistencies Based on Model Operations. In: CAiSE'09–21st Int. Conference on Advanced Information Systems Engineering. (2009)