A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications

Hartmut Ehrig[1], Claudia Ermel[1], and Gabriele Taentzer[2]

[1] Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de, ehrig@cs.tu-berlin.de
[2] Philipps-Universität Marburg, Germany
taentzer@informatik.uni-marburg.de

Abstract. In model-driven engineering, models are primary artifacts and can evolve heavily during their life cycle. Hence, 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, our approach takes abstract syntax structures in model states and operational features into account. Considering the abstract syntax of models as graphs, we define a model revision by a span $G \leftarrow D \rightarrow H$, called graph modification, where G and H are the old and new versions, respectively, and D the common subgraph that remains unchanged. Based on notions of behavioural equivalence and parallel independence of graph modifications, we are able to show a Local-Church-Rosser Theorem for graph modifications. The main goal of the paper is to handle conflicts of graph modifications which may occur in the case of parallel dependent graph modifications. The main result is a general merge construction for graph modifications that resolves all conflicts simultaneously in the sense that for delete-insert conflicts insertion has priority over deletion.

Keywords: graph modification, graph transformation, model versioning, conflict resolution.

1 Introduction

Visual models are primary artifacts in model-driven engineering. Like source code, models may heavily evolve during their life cycle and should put under version control to allow for concurrent modifications of one and the same model by multiple modelers at the same time. When concurrent modifications are allowed, contradicting and inconsistent changes might occur leading to versioning conflicts. 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 visual models, the result is unsatisfactory because the information stemming from abstract syntax structures might be destroyed and associated syntactic and semantic information might get lost.

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 202–216, 2011.

Since the abstract syntax of visual models can be well described by graphs, we consider graph modifications to reason about model evolution. Graph modifications formalize the differences of two graphs before and after a change as a span of injective graph morphisms $G \longleftarrow D \longrightarrow H$ where D is the unchanged part, and we assume wlog. that $D \rightarrow G$ and $D \rightarrow H$ are inclusions. An approach to conflict detection based on graph modifications is described in [10]. We distinguish *operation-based* conflicts where deletion actions are in conflict with insertion actions and *state-based* conflicts where the tentative merge result of two graph modifications is not well-formed wrt. a set of language-specific constraints.

In this paper, we enhance the concepts of [10] by the resolution of operation-based conflicts of graph modifications. First of all, we define behavioural equivalence and parallel independence of graph modifications based on pushout constructions in analogy to algebraic graph transformations [3] and show a Local Church-Rosser Theorem for parallel independent graph modifications. Then we present a merge construction for conflict-free graph modifications and show that the merged graph modification is behavioural equivalent to the parallel composition of the given graph modifications.

The main new idea of this paper is a general merge construction for graph modifications which coincides with the conflict-free merge construction if the graph modifications are parallel independent. Our general merge construction can be applied to conflicting graph modifications in particular. We establish a precise relationship between the behaviour of the given modifications and the merged modification concerning deletion, preservation and creation of edges and nodes. In our main result, we show in which way different conflicts of the given graph modifications are resolved by the merge construction which gives insertion priority over deletion in case of delete-insert conflicts. Note, however, that in general the merge construction has to be processed further by hand, if other choices of conflict resolution are preferred for specific cases. Our running example is a model versioning scenario for statecharts where all conflicts are resolved by the general merge construction.

Structure of the paper: In Section 2, we present the basic concepts of algebraic graph modifications, including behavioural equivalence and a Local Church-Rosser Theorem. A general merge construction is presented and analysed in Section 3, where also the main result concerning conflict resolution is given[1] Related work is discussed in Section 4, and a conclusion including directions for future work is given in Section 5.

2 Graph Modifications: Independence and Behavioural Equivalence

Graph modifications formalize the differences of two graphs before and after a change as a span of injective graph morphisms $G \leftarrow D \rightarrow H$ where D is

[1] The long version of the paper containing full proofs is published as technical report [4].

the unchanged part. This formalization suits well to model differencing where identities of model elements are preserved for each element preserved. We recall the definition of graph modifications from [10] here:

Definition 1 (Graph modification). *Given two graphs G and H, a* graph modification $G \overset{D}{\Longrightarrow} H$ *is a span of injective morphisms* $G \xleftarrow{g} D \xrightarrow{h} H$.

Graph D characterizes an *intermediate graph* where all deletion actions have been performed but nothing has been added yet. Wlog. we can assume that g and h are inclusions, i.e. that D is a subgraph of G and of H. G is called *original graph* and H is called *changed* or *result graph*.

Example 1 (Graph modifications). Consider the following model versioning scenario for statecharts. The abstract syntax of the statechart in Figure 1 (a) is defined by the typed, attributed graph in Figure 1 (b). 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 Figure 1 (b), there are containment edges from superstate S0 to its substates S1 and S2[2]. Note that for simplicity of the presentation we abstract from transition events, guards and actions, as well as from other statechart features, but our technique can also be applied to general statecharts. 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 as node ids) and transitions as directed arcs between state nodes. The compact notation of the statechart in Figure 1 (a) is shown in Figure 1 (c).

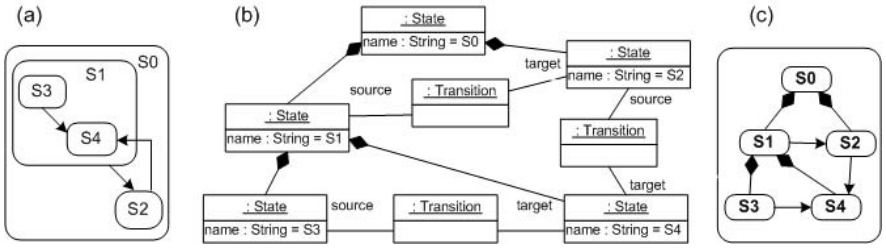


Fig. 1. Sample statechart: concrete syntax (a), abstract syntax graph (b), and compact notation (c)

In our model versioning scenario, two users check out the statechart shown in Figure 1 and change it in two different ways. User 1 performs a refactoring operation on it. She moves state S3 up in the state hierarchy (cf. Figure 2). User 2 deletes state S3 together with its adjacent transition to state S4.

Obviously, conflicts occur when these users try to check in their changes: state S3 is deleted by user 2 but is moved to another container by user 1.

[2] In contrast to UML state machines, we distinguish edges that present containment links by composition decorators.

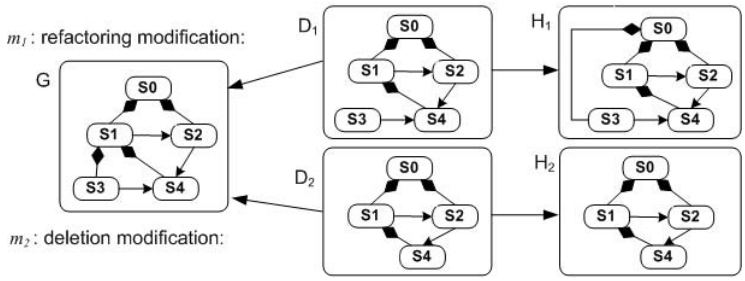


Fig. 2. Graph modifications m_1 (refactoring) and m_2 (deletion)

In this section, we study relations between different graph modifications based on category theory. Due to this general formal setting, we can use different kinds of graphs like labelled, typed or typed attributed graphs (see [3] for more details). At first, we consider the sequential and parallel composition of two graph modifications. Our intention is that graph modifications are closed under composition. Given the sequential composition of two graph modififcations $G \overset{D_1}{\Longrightarrow} H_1$ and $H_1 \overset{D_2}{\Longrightarrow} H_2$, the resulting modification obviously has G as original and H_2 as changed graph. But how does their intermediate graph D should look like? The idea is to construct D as intersection graph of D_1 and D_2 embedded in H_1. This is exactly realized by a pullback construction. The parallel composition of two graph modifications means to perform both of them independently of each other by componentwise disjoint union. This corresponds to coproduct constructions $G_1 + G_2, D_1 + D_2$ and $H_1 + H_2$ on original, intermediate and changed graphs.

Definition 2 (Composition of graph modifications). *Given two graph modifications* $G \overset{D_1}{\Longrightarrow} H_1 = (G \leftarrow D_1 \rightarrow H_1)$ *and* $H_1 \overset{D_2}{\Longrightarrow} H_2 = (H_1 \leftarrow D_2 \rightarrow H_2)$, *the* sequential composition *of* $G \overset{D_1}{\Longrightarrow} H_1$ *and* $H_1 \overset{D_2}{\Longrightarrow} H_2$, *written* $(G \leftarrow D_1 \rightarrow H_1) * (H_1 \leftarrow D_2 \rightarrow H_2)$ *is given by* $G \overset{D}{\Longrightarrow} H_2 = (G \leftarrow D \rightarrow H_2)$ *via the pullback construction*

$$\begin{array}{ccccccc} G & \longleftarrow & D_1 & \longrightarrow & H_1 & \longleftarrow & D_2 \longrightarrow H_2. \\ & & & \nwarrow & (PB) & \nearrow & \\ & & & & D & & \end{array}$$

The parallel composition *of* $G_1 \overset{D_1}{\Longrightarrow} H_1$ *and* $G_2 \overset{D_2}{\Longrightarrow} H_2$ *is given by coproduct construction:* $G_1 + G_2 \overset{D_1+D_2}{\Longrightarrow} H_1 + H_2 = (G_1 + G_2 \leftarrow D_1 + D_2 \rightarrow H_1 + H_2)$.

The differences between the original and the intermediate graph as well as between the intermediate and the changed graph define the behaviour of a graph modification. The same behaviour can be observed in graphs with more or less context. Therefore, we define the behavioural equivalence of two graph modifications as follows: Starting with two modifications $m_i = (G_i \overset{D_i}{\Longrightarrow} H_i)$ $(i = 1, 2)$, we look for a third graph modification $G \overset{D}{\Longrightarrow} H$ modeling the same changes with so little context that it can be embedded in m_1 and m_2. A behaviourally equivalent embedding of graph modifications can be characterized best by two

pushouts as shown in Definition 3, since the construction of a pushout ensures that G_i are exactly the union graphs of G and D_i overlapping in D. Analogously, H_i are exactly the union graphs of H and D_i overlapping in D[3].

Definition 3 (Behavioural Equivalence of Graph Modifications)
Two graph modifications $G_i \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *are called* behaviourally equivalent *if there is a span* $(G \leftarrow D \rightarrow H)$ *and PO-span morphisms from* $(G \leftarrow D \rightarrow H)$ *to* $(G_i \leftarrow D_i \rightarrow H_i)$, $(i = 1, 2)$, *i.e. we get four pushouts in the diagram to the right.*

$$\begin{array}{ccccc} G_1 & \longleftarrow & D_1 & \longrightarrow & H_1 \\ \uparrow & (PO) & \uparrow & (PO) & \uparrow \\ G & \longleftarrow & D & \longrightarrow & H \\ \downarrow & (PO) & \downarrow & (PO) & \downarrow \\ G_2 & \longleftarrow & D_2 & \longrightarrow & H_2 \end{array}$$

Example 2. Figure 3 shows two behaviourally equivalent graph modifications where the upper one is the refactoring modification m_1 from Figure 2. The span $(G \leftarrow D \rightarrow H)$ shows the same changes as in m_1 and m_2, but in less context.

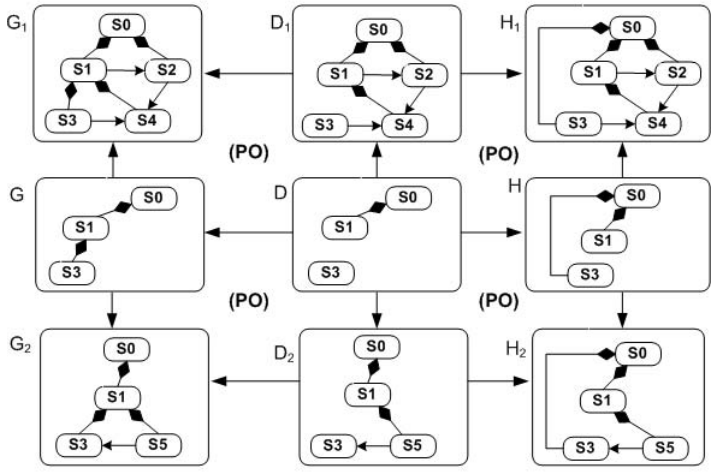


Fig. 3. Graph modifications m_1 and m_2 are behaviourally equivalent

We want to consider graph modifications to be parallel independent if they do not interfere with each other, i.e. one modification does not delete a graph element the other one needs to perform its changes. While nodes can always be added to a graph independent of its form, this is not true for edges. An edge can only be added if it has a source and a target node. Thus parallel independence means more concretely that one modification does not delete a node that is

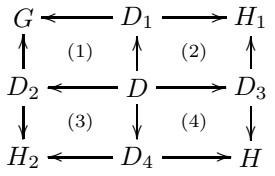
[3] In the framework of algebraic graph transformations [3], we may also consider graph modification $G \overset{D}{\Longrightarrow} H$ as a graph rule r which is applied to two different graphs G_1 and G_2. Since the same rule is applied, graph transformations $G_i \overset{r}{\Longrightarrow} H_i$ $(i = 1, 2)$ would be behaviourally equivalent.

supposed to be the source or target node of an edge to be added by the other modification. Moreover, both graph modifications could delete the same graph elements. It is debatable whether the common deletion of elements can still be considered as parallel independent or not. Since we consider parallel independent modifications to be performable in any order, common deletions are not allowed. Once modification m_1 has deleted a graph element, it cannot be deleted again by modification m_2[4].

This kind of parallel independence is characterized by Definition 4 as follows: At first, we compute the intersection D of D_1 and D_2 in G by constructing a pullback. Since common deletions are not allowed, there has to be at least one modification for each graph element which preserves it. Thus, D_1 glued with D_2 via D has to lead to G, which corresponds to pushout (1). Next, considering D included in D_1 included in H_1 we look for some kind of graph difference. We want to identify those graph elements of H_1 that are not already in D_1 and have to be added by D_3 such that both overlap in D, i.e. H_1 becomes the pushout object of $D \to D_1$ and $D \to D_3$. In this case, D_3 with $D \to D_3 \to H_1$ is called pushout complement of $D \to D_1 \to H_1$ (see [3] for pushout and pushout complement constructions). Analogously, D_4 is the difference of H_2 and D_2 modulo D. Finally, both differences D_3 and D_4 are glued via D resulting in H. According to Proposition 1, both modifications may occur in any order, such that Definition 4 reflects precisely our intention of parallel independence.

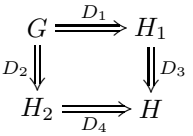
Definition 4 (Parallel Independence of Graph Modifications)

Two graph modifications $(G \leftarrow D_i \rightarrow H_i)$, $(i = 1, 2)$ *are called* parallel independent *if we have the four pushouts in the diagram to the right, where* (1) *can be constructed as pullback,* (2) *and* (3) *as pushout-complements and* (4) *as pushout.*



Proposition 1 (Local Church-Rosser for Graph Modifications).

Given parallel independent graph modifications $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$. *Then, there exists* H *and graph modifications* $H_1 \overset{D_3}{\Longrightarrow} H$, $H_2 \overset{D_4}{\Longrightarrow} H$ *which are behaviourally equivalent to* $G \overset{D_2}{\Longrightarrow} H_2$, $G \overset{D_1}{\Longrightarrow} H_1$, *respectively.*



Proof. Given parallel independent graph modifications $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$, we have pushouts $(1) - (4)$ by Definition 4 leading to $H_1 \overset{D_3}{\Longrightarrow} H$ and $H_2 \overset{D_4}{\Longrightarrow} H$ with behavioural equivalence according to Definition 3. □

Example 3. Figure 4 shows two parallel independent graph modifications where $m_1 = (G \leftarrow D_1 \rightarrow H_1)$ is the refactoring modification from Figure 2, and $m_2 = (G \leftarrow D_2 \rightarrow H_2)$ deletes the transition from `S2` to `S4` and adds a new state named `S5`. We have four pushouts, thus both graph modifications may be performed in any order, yielding in both cases the same result H.

[4] However, we will see that modifications with common deletions still can be merged.

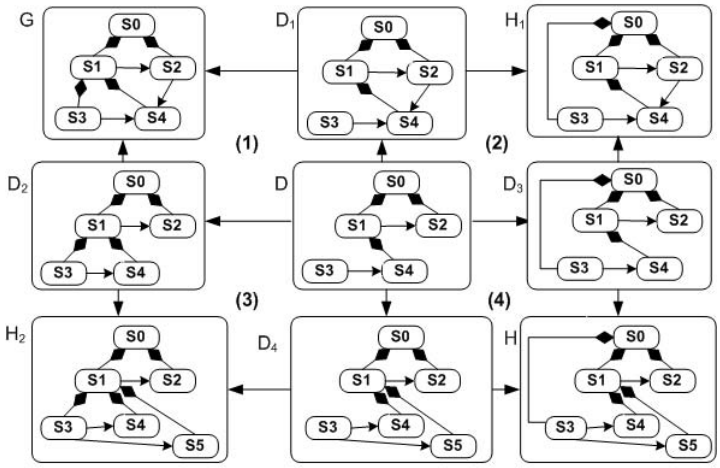


Fig. 4. Parallel independent graph modifications

In the case that two graph modifications are parallel independent, they are called *conflict-free* and can be merged to one *merged graph modification* that realizes both original graph modifications simultaneously. Note that the merge construction in Definition 5 corresponds to the construction in [10].

Definition 5 (Merging Conflict-Free Graph Modifications). *Given parallel independent graph modifications* $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$. *Then, graph modification* $G \overset{D}{\Longrightarrow} H$ *given by* $G \leftarrow D \rightarrow H$ *defined by the diagonals of pushouts* (1), (4) *in Definition 4 is called* merged graph modification *of* $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$.

In [4], we show that in case of parallel independence of $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$, the merged graph modification $G \overset{D}{\Longrightarrow} H$ is behaviourally equivalent to the parallel composition $G + G \overset{D_1+D_2}{\Longrightarrow} H_1 + H_2$ of the original graph modifications $G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$. This result confirms our intuition that the merged graph modification $G \overset{D}{\Longrightarrow} H$ realizes both graph modifications simultaneously. Moreover, it is equal to the sequential compositions of $G \overset{D_1}{\Longrightarrow} H_1 \overset{D_3}{\Longrightarrow} H$ and $G \overset{D_2}{\Longrightarrow} H_2 \overset{D_4}{\Longrightarrow} H$ in Definition 4.

Example 4. The merged graph modification of the two parallel independent graph modifications $m_1 = (G \leftarrow D_1 \rightarrow H_1)$ and $m_2 = (G \leftarrow D_2 \rightarrow H_2)$ in Figure 4 is given by $m = (G \leftarrow D \rightarrow H)$. Obviously, m realizes both graph modifications m_1 and m_2 simultaneously and is shown in [4] to be behaviourally equivalent to their parallel composition $G + G \overset{D_1+D_2}{\Longrightarrow} H_1 + H_2$ and equal to the sequential compositions $G \overset{D_1}{\Longrightarrow} H_1 \overset{D_3}{\Longrightarrow} H$ and $G \overset{D_2}{\Longrightarrow} H_2 \overset{D_4}{\Longrightarrow} H$.

3 Conflict Resolution

If two graph modifications have conflicts, a merge construction according to Definition 5 is not possible any more. In this section, we propose a general merge construction that resolves conflicts by giving *insertion* priority over *deletion* in case of delete-insert conflicts. The result is a merged graph modification where the changes of both original graph modifications are realized as far as possible. We state the properties of the general merge construction and show that the merge construction for the conflict-free case is a special case of the general merge construction.

Definition 6 (Conflicts of Graph Modifications)

1. *Two modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *are* conflict-free *if they are parallel independent (i.e. we have four pushouts according to Definition 4).*
2. *They are in* conflict *if they are not parallel independent.*
3. *They are in* delete-delete conflict *if* $\exists x \in (G \backslash D_1) \cap (G \backslash D_2)$.
4. (m_1, m_2) *are in* delete-insert conflict *if*

$$\exists \text{ edge } e \in H_2 \backslash D_2 \text{ with } s(e) \in D_2 \cap (G \backslash D_1) \text{ or } t(e) \in D_2 \cap (G \backslash D_1).$$

Example 5. Consider the graph modifications $m_1 = G \leftarrow D_1 \rightarrow H_1$ and $m_2 = G \leftarrow D_2 \rightarrow H_2$ in Figure 2. (m_2, m_1) are in *delete-insert* conflict because m_2 deletes node `S3` which is needed by m_1 for the insertion of an edge. Moreover, m_1 and m_2 are in *delete-delete* conflict because the edge from `S1` to `S3` is deleted by both m_1 and m_2. (m_1, m_2) are not in delete-insert conflict.

If two modifications m_1 and m_2 are in conflict, then at least one conflict occurs which can be of the following kinds: (1) both modifications delete the same graph element, (2) m_1 deletes a node which shall be source or target of a new edge inserted by m_2, and (3) m_2 deletes a node which shall be source or target of a new edge inserted by m_3. Of course, several conflicts may occur simultaneously. In fact, all three conflict situations may occur independently of each other. For example, (m_1, m_2) may be in delete-delete conflict, but not in delete-insert conflict, or vice versa. [5]

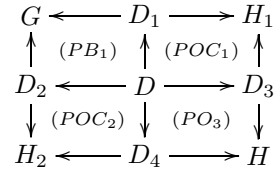
Theorem 1 characterizes the kinds of conflicts that parallel dependent graph modifications may have.

Theorem 1 (Characterization of Conflicts of Graph Modifications). *Given* $m_i = (G \overset{D_i}{\Longrightarrow} H_i)$ $(i = 1, 2)$, *then* (m_1, m_2) *are in conflict iff*

1. (m_1, m_2) *are in delete-delete conflict, or*
2. (m_1, m_2) *are in delete-insert conflict, or*
3. (m_2, m_1) *are in delete-insert conflict.*

[5] In the worst case, we may have all kinds of conflicts simultaneously.

Proof Idea. Parallel independence of (m_1, m_2) is equivalent to the fact that (PB_1) is also pushout, and the pushout complements (POC_1) and (POC_2) exist, such that pushout (PO_3) can be constructed. By negation, statements *1. - 3.* are equivalent to *4. - 6.*, respectively:



4. (PB_1) is not a pushout, i.e. $D_1 \to G \leftarrow D_2$ is not jointly surjective.
5. The dangling condition for $D \to D_2 \to H_2$ is not satisfied.
6. The dangling condition for $D \to D_1 \to H_1$ is not satisfied.

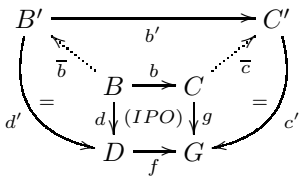
The *dangling condition* mentioned in statements *5. - 6.* is the one known from DPO graph transformation [3]. It is satisfied by inclusions $D \to D_i \to H_i$ $(i = 1, 2)$, if $\forall e \in H_i \backslash D_i : (s(e) \in D_i \implies s(e) \in D) \wedge (t(e) \in D_i \implies t(e) \in D)$. $s(e)$ and $t(e)$ are called *dangling points*. If the dangling condition is satisfied by $D \to D_i \to H_i$, the pushout complement (POC_i) can be constructed. □

For delete-insert conflicts, our preferred resolution strategy is to preserve the nodes in the merged graph modification that are needed to realize the insertion of edges. If deletion is preferred instead, it has to be done manually after the automatic construction of the merged graph modification, supported by visual conflict indication. Ideally, deletion is done such that predefined meta-model constraints are fulfilled afterwards (see conclusion).

In the following, we define a general merge construction yielding the desired merged graph modification for two given graph modifications with conflicts. In the special case that we have parallel independent graph modifications, it coincides with the conflict-free merge construction in Definition 5.

For the general merge construction, we need so-called *initial pushouts*. In a nutshell, an initial pushout over a graph morphism $f : D \to G$ extracts a minimal graph morphism $b : B \to C$ where the context C contains all non-mapped parts of G, and the boundary B consists of those nodes in D that are used for edge insertion (see [3]).

Definition 7 (Initial pushout). *Let $f\colon D \to G$ be a graph morphism, an* initial pushout *over f consists of graph morphisms $g\colon C \to G$, $b\colon B \to C$, and injective $d\colon B \to D$ such that f and g are a pushout over b and d. For every other pushout over f consisting of $c'\colon C' \to G$, $b'\colon B' \to C'$, and injective $d'\colon B' \to D$, there are unique graph morphisms $\overline{b}\colon B \to B'$ and $\overline{c}\colon C \to C'$ such that $c' \circ \overline{c} = g$ and $d' \circ \overline{b} = d$. Moreover, it is required that $(\overline{c}, b')$ is a pushout over $(b, \overline{b})$.*



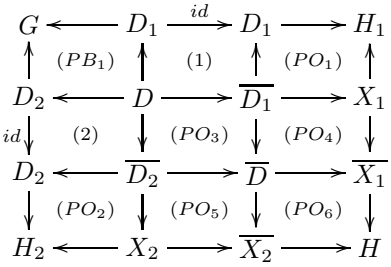
Note that for graph morphisms, there is a canonical construction for initial pushouts [3].

Example 6 (Initial pushout). In the upper right corner of our sample merge construction diagram in Figure 5 below, the initial pushout IPO_1 over the morphism

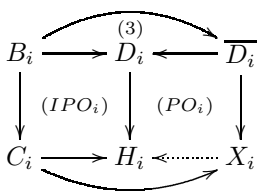
$D_1 \to H_1$ of graph modification m_1 in Figure 2 is shown. Obviously, the morphism $B_1 \to C_1$ contains in a minimal context the insertion of the containment edge from S0 to S3.

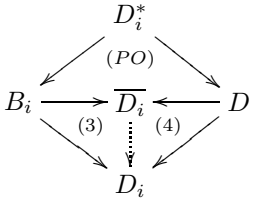
Now, we are ready to present our general merge construction for graph modifications (see Definition 8). Analogously, to the merging of conflict-free graph modifications we start with constructing the intersection D of the intermediate graphs D_1 and D_2. In case of delete-insert conflicts where a node is supposed to be deleted by one modification and used as source or target by the other modification, D is too small, i.e. does not contain such nodes. Therefore, we look for a construction which enlarges D to the intermediate graph for the merged modification where insertion is prior to deletion: At first, we identify all these insertions in modifications 1 and 2. This is done by initial pushout construction (as described above) leading to $B_i \to C_i (i = 1, 2)$. By constructing first the intersection D_i^* of B_i and D in D_i and thereafter the union $\overline{D_i}$ of B_i and D via D_i^*, graph D is extended by exactly those graph elements in B_i needed for insertion later on resulting in $\overline{D_i}$. After having constructed these extended intermediate graphs $\overline{D_1}$ and $\overline{D_2}$, they have to be glued to result in the intermediate graph $\overline{D}$ of the merged graph modification. Thereafter, the insertions identified by $B_i \to C_i (i = 1, 2)$ can be transferred to $\overline{D_i} \to X_i (i = 1, 2)$ first and to $\overline{D} \to \overline{X_i} (i = 1, 2)$ thereafter. Finally, they are combined by gluing $\overline{X_1}$ and $\overline{X_2}$ via $\overline{D}$ yielding result graph H. Since $\overline{D}$ is D extended by graph elements which are not to be deleted, $\overline{D}$ can be embedded into G and thus, can function as intermediate graph for the merged graph modification $G \leftarrow \overline{D} \to H$.

Definition 8 (Merged Graph Modification in General). *Given two graph modifications $G \leftarrow D_1 \to H_1$ and $G \leftarrow D_2 \to H_2$. We construct their* merged graph modification $G \leftarrow \overline{D} \to H$ *in 6 steps, leading to the following* general merge construction *diagram:*

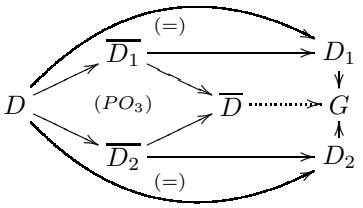


1. *Construct D by pullback (PB_1) of $D_1 \to G \leftarrow D_2$.*
2. *Construct initial pushouts (IPO_i) over $D_i \to H_i$ for $i = 1, 2$:*





3. *Construct* D_i^* *as a pullback of* $B_i \to D_i \leftarrow D$ *and* $\overline{D_i}$ *as pushout of* $B_i \leftarrow D_i^* \to D$ *with induced morphism* $\overline{D_i} \to D_i$ *with* $B_i \to \overline{D_i} \to D_i = B_i \to D_i$ (3) *and* $D \to \overline{D_i} \to D_i = D \to D_i$ (4) *for* $(i = 1, 2)$.
4. *Construct pushout* $\overline{D_i} \to X_i \leftarrow C_i$ *of* $\overline{D_i} \leftarrow B_i \to C_i$ $(i = 1, 2)$, *leading by* (3) *to induced morphism* $X_i \to H_i$ *and pushout* (PO_i) $(i = 1, 2)$ *by pushout decomposition. Moreover,* (4) *implies commutativity of* (1) *and* (2) *for* $(i = 1, 2)$.
5. *Now we are able to construct pushouts* (PO_3), (PO_4), (PO_5) *and* (PO_6) *one after the other.*
6. *Finally, we obtain the* merged graph modification $(G \leftarrow \overline{D} \to H)$, *where* $\overline{D} \to H$ *is defined by composition in* (PO_6), *and* $\overline{D} \to G$ *is uniquely defined as induced morphism using pushout* (PO_3).



Remark 1. If the modifications $m_i = (G \leftarrow D_i \to H_i)$ $(i = 1, 2)$ are parallel independent, then the pullback (PB_1) is a pushout and $\overline{D_1} = D = \overline{D_2} = \overline{D}$. In this case, the general merged modification $m = (G \leftarrow \overline{D} \to H)$ is equal up to isomorphism to the merged graph modification in the conflict-free case in Definition 5. If $m_i = (G \leftarrow D_i \to H_i)$ $(i = 1, 2)$ are in delete-delete conflict, then the merged graph modification deletes the items that are deleted by both m_1 and m_2 since these items are not in D and hence not in $\overline{D}$.

Example 7. We construct the merged graph modification for graph modifications $m_1 = G \leftarrow D_1 \to H_1$ and $m_2 = G \leftarrow D_2 \to H_2$ in Figure 2. The construction diagram is shown in Figure 5.

According to step 3 in Definition 8, $\overline{D_1}$ has to be constructed as pushout of $B_1 \leftarrow D_1^* \to D$. D_1^* is the pullback of $B_1 \to D_1 \leftarrow D$, hence D_1^* consists just of the single node `S0` . Since B_1 contains two single nodes, `S0` and `S3` , we get as result of step 3 graph $\overline{D_1}$ which is similar to D but contains additionally node `S3`. Since (m_1, m_2) are not in delete-insert conflict, $\overline{D_2} = D$. All remaining squares are constructed as pushouts.

Note that the resulting merged graph modification $G \leftarrow \overline{D} \to H$ preserves node `S3` because this node is deleted in m_2 although it is used for inserting a new edge in m_1 (resolution of the delete-insert conflict). The edge from `S1` to `S3` is deleted by the merged graph modification as it is deleted by both m_1 and m_2 (resolution of the delete-delete conflict). All graph objects created by either m_1 or m_2 are created also by the merged graph modification. Note that square (2) is a pushout in this example since (m_1, m_2) are not in delete-insert conflict.

The following theorem states that the modification resulting from the general merge construction specifies the intended semantics resolving delete-insert conflicts by preferring insertion over deletion:

Theorem 2 (Behaviour Compatibility of the General Merge Construction). *Given graph modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *with merged graph*

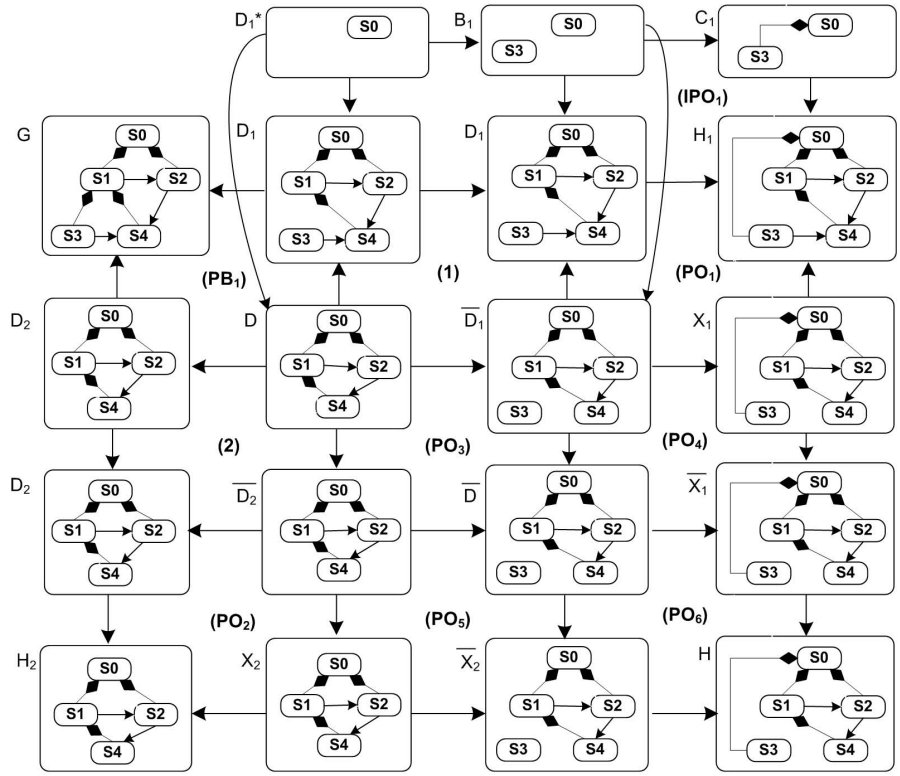


Fig. 5. General merge construction for conflicting graph modifications m_1 and m_2

modification $m = G \overset{\overline{D}}{\Longrightarrow} H = (G \leftarrow \overline{D} \rightarrow H)$ *in the sense of Definition 8. We use the following terminology for* m *(and similarly for for* m_1, m_2*):*

$x \in G$ *preserved by* $m \iff x \in \overline{D}$,
$x \in G$ *deleted by* $m \iff x \in G \backslash \overline{D}$,
$x \in H$ *created by* $m \iff x \in H \backslash \overline{D}$.

Then, m *is behaviour compatible with* m_1 *and* m_2 *in the following sense:*

1. Preservation: $x \in G$ *preserved by* m_1 *and* $m_2 \Longrightarrow x \in G$ *preserved by* m
   $\Longrightarrow x \in G$ *preserved by* m_1 *or* m_2
2. Deletion: $x \in G$ *deleted by* m_1 *and* $m_2 \Longrightarrow x \in G$ *deleted by* m
   $\Longrightarrow x \in G$ *deleted by* m_1 *or* m_2
3. Preservation and Deletion: $x \in G$ *preserved by* m_1 *and* $x \in G$ *deleted by* m_2
   $\Longrightarrow x \in G$ *preserved by* m, *if* $x \in \overline{D_1}$[6]
   $x \in G$ *deleted by* m, *if* $x \notin \overline{D_1}$[7]
   (similar for $m_1, m_2, \overline{D_1}$ *replaced by* $m_2, m_1, \overline{D_2}$*)*

[6] In this case, x is a node needed as source or target for an edge inserted by m_1.
[7] In this case, x is not needed for edge insertion by m_1.

4. Creation: $x \in H_1$ *created by* m_1 *or* $x \in H_2$ *created by* m_2
$\Longleftrightarrow x \in H$ *created by* m

Proof Idea. The preservation, deletion and creation results follow from the pushout properties of $\overline{D}$, the pushout complement properties of $\overline{D_1}$, $\overline{D_2}$ and the fact that $\overline{D_1}$ is pullback in the diagrams (PO_1) and (PO_4) (and analogously for $\overline{D_2}$).

Theorem 3 characterizes the three forms of conflict resolution which may occur.

Theorem 3 (Conflict Resolution by General Merge Construction). *Given graph modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *that are in conflict. The merge construction* $m = (G \leftarrow \overline{D} \rightarrow H)$ *resolves the conflicts in the following way:*

1. *If* (m_1, m_2) *are in delete-delete conflict, with both* m_1 *and* m_2 *deleting* $x \in G$, *then* x *is deleted by* m.
2. *If* (m_1, m_2) *are in delete-insert conflict, there is an edge* e_2 *created by* m_2 *with* $x = s(e_2)$ *or* $x = t(e_2)$ *preserved by* m_2, *but deleted by* m_1. *Then* x *is preserved by* m.
3. *If* (m_2, m_1) *are in delete-insert conflict, there is an edge* e_1 *created by* m_1 *with* $x = s(e_1)$ *or* $x = t(e_1)$ *preserved by* m_1, *but deleted by* m_2. *Then* x *is preserved by* m.

Proof Idea. The resolution of delete-delete conflicts follows from the deletion property, and the resolution of delete-insert conflicts follows from the preservation-deletion property of the general merge construction in Theorem 2.

4 Related Work

First of all, we have to clarify that model merging differs from merging of model modifications. Model merging as presented e.g. in [7,9] is concerned with a set of models and their inter-relations expressed by binary relations. In contrast, merging of model modifications takes change operations into account. Merging of model modifications usually means that non-conflicting parts are merged automatically, while conflicts have to be resolved manually. In the literature, different resolution strategies which allow at least semi-automatic resolution are proposed. A survey on model versioning approaches and especially on conflict resolution strategies is given in [1].

A category-theoretical approach formalizing model versioning is given in [8]. Similar to our approach, modifications are considered as spans of morphisms to describe a partial mapping of models. Merging of model changes is defined by pushout constructions. However, conflict resolution is not yet covered by this approach in a formal way. A category theory-based approach for model versioning in-the-large is given in [2]. However, this approach is not concerned with formalizing conflict resolution strategies. A set-theoretic definition of EMF model merging is presented in [12], but conflicts are solved by the user and not automatically.

In [5] the applied operations are identified first and grouped into parallel independent subsequences then. Conflicts can be resolved by either (1) discarding complete subsequences, (2) combining conflicting operations in an appropriate way, or (3) modifying one or both operations. The choice of conflict resolution is made by the modeler. These conflict resolution strategies have not been formalized. The intended semantics is not directly investigated but the focus is laid on the advantage of identifying compound change operations instead of elementary ones. In contrast, we propose a semi-automatic procedure where at first, an automatic merge construction step gives insertion priority over deletion in case of delete-insert conflicts. If other choices are preferred, the user may perform deletions manually in a succeeding step.

Automatic merge results may not always solve conflicts adequately, especially state-based conflicts or inconsistencies may still exist or arise by the merge construction. Resolution strategies such as resolution rules presented in [6] are intended to solve state-based conflicts or inconsistencies. They can be applied in follow-up graph transformations after the general conflict resolution procedure produced a tentative merge result.

5 Conclusions and Future Work

In this paper, we have formalized a conflict resolution strategy for operation-based conflicts based on graph modifications. Our main result is a general merge construction for conflicting graph modifications. The merge construction realizes a resolution strategy giving insertion priority over deletion in case of delete-insert conflicts to get a merged graph modification result containing as much information as possible. We establish a precise relationship between the behaviour of the given graph modifications and the merged modification concerning deletion, preservation and creation of graph items. In particular, our general merge construction coincides with the conflict-free merge construction if the graph modifications are parallel independent. We show how different kinds of conflicts of given graph modifications are resolved by our automatic resolution strategy. It is up to an additional manual graph modification step to perform deletions that are preferred over insertions.

In [10], we presented two kinds of conflicts which can be detected based on graph modification: *operation-based* and *state-based* conflicts. Hence, in future work, our strategy for solving operation-based conflicts shall be extended by resolving also state-based conflicts. Here, repair actions should be provided to be applied manually by the modeler. Their applications would lead to additional graph modifications optimizing the merged graph modification obtained so far. For the specification of repair actions in this setting, the work by Mens et al. in [6] could be taken into account.

With regard to tool support, our graph transformation environment AGG [11] supports conflict analysis for graph rules and graph modifications. We plan to implement also the check of behavioural equivalence and the general merge

construction for graph modifications in near future. This proof-of-concept implementation could function as blueprint for implementing our new resolution strategy in emerging model versioning tools.

References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. IJWIS 5(3), 271–304 (2009)
2. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: Proc. of Workshop on Comparison and Versioning of Software Models (CVSM 2009), pp. 7–12. IEEE Computer Society, Los Alamitos (2009)
3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. In: EATCS Monographs in Theor. Comp. Science. Springer, Heidelberg (2006)
4. Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications: Extended version. Tech. rep., TU Berlin (to appear, 2011)
5. Küster, J.M., Gerth, C., Engels, G.: Dependent and conflicting change operations of process models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 158–173. Springer, Heidelberg (2009)
6. Mens, T., van der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
7. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) VLDB 2003. LNCS, vol. 2944, pp. 826–873. Springer, Heidelberg (2004)
8. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A category-theoretical approach to the formalisation of version control in MDE. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 64–78. Springer, Heidelberg (2009)
9. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: Proc. IEEE Int. Conf. on Requirements Engineering, pp. 221–230. IEEE, Los Alamitos (2007)
10. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 171–186. Springer, Heidelberg (2010)
11. TFS-Group, TU Berlin: AGG (2009), `http://tfs.cs.tu-berlin.de/agg`
12. Westfechtel, B.: A formal approach to three-way merging of EMF models. In: Proc. Workshop on Model Comparison in Practice (IWMCP), pp. 31–41. ACM, New York (2010)