Softw Syst Model
DOI 10.1007/s10270-012-0248-x

THEME SECTION PAPER

A fundamental approach to model versioning based on graph modifications: from theory to implementation

Gabriele Taentzer · Claudia Ermel · Philip Langer · Manuel Wimmer

Received: 12 April 2011 / Revised: 14 March 2012 / Accepted: 5 April 2012

Abstract In model-driven engineering, models are primary artifacts that can evolve heavily during their life cycle. Therefore, versioning of models is a key technique to be offered by integrated development environments for model-driven engineering. In contrast to text-based versioning systems, we present an approach that takes model structures and their changes over time into account. Considering model structures as graphs, we define a fundamental approach where model revisions are considered as graph modifications consisting of delete and insert actions. Two different kinds of conflict detection are presented: (1) the check for operation-based conflicts between different graph modifications, and (2) the check for state-based conflicts on merged graph modifications. For the merging of graph modifications, a two-phase approach is proposed: First, operational conflicts are temporarily resolved by always giving insertion priority over deletion to keep as much information as possible. Thereafter, this tentative merge result is the basis for manual conflict resolution as well as for the application of repair actions that resolve state-based conflicts. If preferred by the user, giving deletion priority over insertion might be one solution. The fundamental concepts are illustrated by versioning scenarios for simplified statecharts. Furthermore, we show an implementation of this fundamental approach to model versioning based on the Eclipse Modeling Framework as technical space.

Keywords Model versioning · Graph modification · Conflict detection · 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 be 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 text files 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 model structures is certainly shown in an inadequate way such that associated syntactic and semantic information cannot be recognized.

To tackle this problem, dedicated model versioning systems have been proposed [9,30,42,50]. 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 fundamental concepts of potentially occurring kinds of conflicts and their resolution is indispensable,

Communicated by Dr. Andy Schürr and Arend Rensink.

M. Wimmer's work has been partially funded by the Austrian Science Fund (FWF) under Grant J 3159-N23.

G. Taentzer
Philipps-Universität Marburg, Marburg, Germany
e-mail: taentzer@mathematik.uni-marburg.de

C. Ermel
Technische Universität Berlin, Berlin, Germany
e-mail: claudia.ermel@tu-berlin.de

P. Langer (✉) · M. Wimmer
Vienna University of Technology, Vienna, Austria
e-mail: langer@big.tuwien.ac.at

M. Wimmer
e-mail: wimmer@big.tuwien.ac.at

Published online: 25 April 2012

but yet missing. Throughout this paper, we consider the construction of model differences between an original model and its revisions. Thereafter, two model differences w.r.t. the same original model are selected and the so-called three-way model merge is computed.

Model structures, especially for visual models, are well described by graphs, since their elements do not have a natural ordering in general. Considering, e.g., class model, although there might be elaborated generalization relations, their use relations do not form trees, but graphs in general. Based on the definition of model structures by graphs, we consider graph modifications to reason about model evolution. A graph modification formalizes the difference of two graphs before and after a change such that preserved graph items can be identified. However, the order of model changes is not tracked. This basic setting is well suited to reason about model versioning independent of any technical space. For efficient implementation of the considered concepts it might be worthwhile to consider more specific structures than graphs.

In [51], we introduce our approach to conflict detection based on graph modifications. *Operation-based* conflicts, where deletion actions are in conflict with insertion actions, are distinguished from *state-based* conflicts where the tentative merge result of two graph modifications is not well formed w.r.t. a set of consistency constraints. In this paper, we enhance the conflict detection presented in [51] by a resolution of operation-based conflicts in graph modifications. We present a semi-automatic merge construction for graph modifications which tentatively resolves delete–insert conflicts by giving priority to insertion. This resolution strategy keeps as much information as possible. In [23], this strategy is formally defined and it is shown that the constructed merge result is compatible with the intended behavior and resolves all conflicts reported. However, a conflict resolution by insertion is not always the resolution preferred by the user. Therefore, and in case of additional state-based conflicts, the tentative merge result can be processed further by the application of repair actions. Figure 1 summarizes the merge process of the proposed approach at a glance.

Graphs, graph operations, and graph modifications are well suited to provide a fundamental understanding of model versioning problems independent of specific technical spaces. In addition, we choose one technical space for modeling, i.e., the Eclipse Modeling Framework (EMF) [20], and present how the fundamental graph-based concepts can be implemented in this context. After recalling all features and peculiarities of EMF, we provide a detailed insight in the prototypical implementation of the EMF-based model versioning system AMOR[1] [9] as well as its relation to the fundamental concepts for model versioning.

[1] http://modelversioning.org.

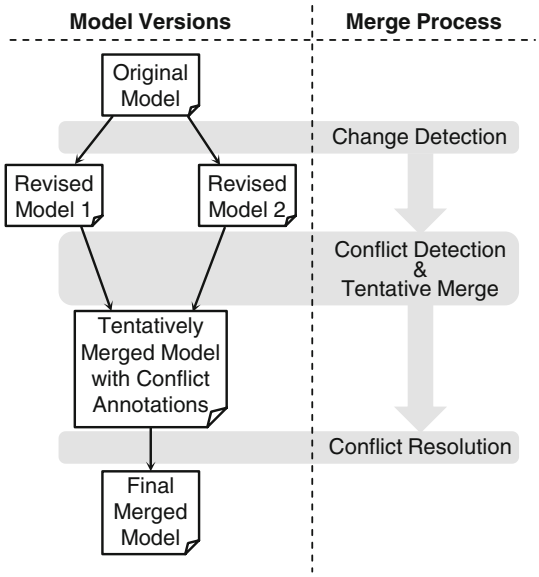


Fig. 1 Merge process at a glance

All definitions and constructions are illustrated at a running example being a model versioning scenario for simple statecharts.

Structure of the paper In Sect. 2, we present the basic concepts of graphs and graph modifications. We define operation-based conflicts in Sect. 3. A tentative merge construction realizing a pre-defined resolution of operation-based conflicts is presented and analyzed in Sect. 4. In this strategy, insertion is given priority over deletion in case of operation-based conflicts. The detection and resolution of state-based conflicts are treated in Sect. 5 and Sect. 6, respectively. In Sect. 7, we discuss how graphs are related to EMF models. Section 8 is concerned with obtaining and representing differences between EMF models and with the alignment of these techniques with graph modifications. The obtained differences are the prerequisites for realizing operation-based conflict detection which is the focus of Sect. 9. In Sect. 10, we present how two concurrently performed modifications of an original EMF model are merged to detect state-based conflicts in EMF models as discussed in Sect. 11. Related work is discussed in Sect. 12, and a conclusion including directions for future work is given in Sect. 13.

2 Graph modifications: a difference model for graphs

Models can differ in various aspects: structure, names, model element identities, and the order of model elements in collections. We focus on models where all elements have identities and elements are not ordered in collections. Furthermore, model changes keep identities of preserved model elements. Thus, model differences can be concerned with model structures as well as element names and their attribute values.

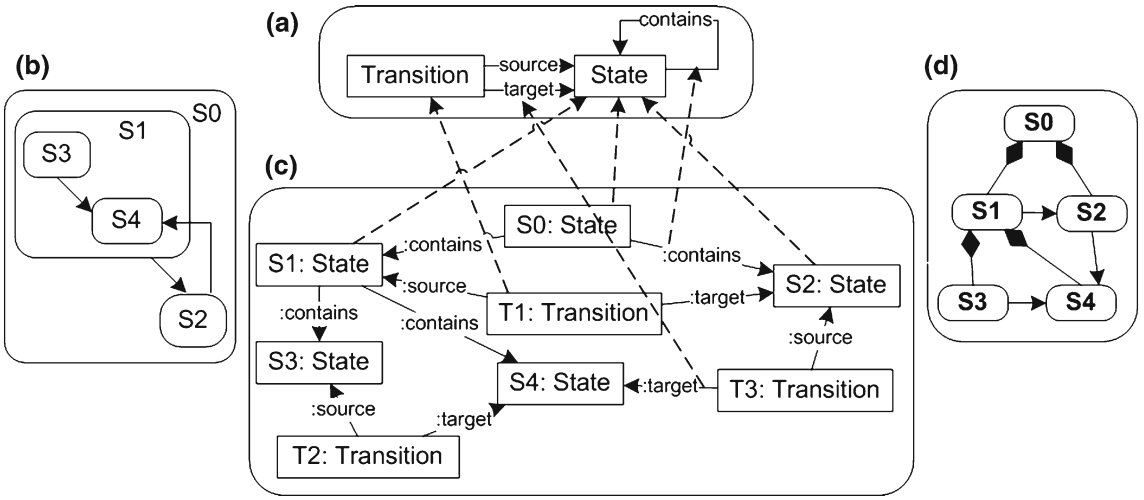


Fig. 2 Statechart type graph (**a**) and sample statechart in concrete syntax (**b**), as abstract syntax graph (**c**), and in compact notation (**d**)

Throughout this paper, we describe the underlying structure of a model by a graph. Graphs are a natural way to represent the underlying structure of models being highly linked structures. While some kinds of models such as well-structured activity diagrams expose tree-like structures, this is not true for models in general. To capture all important information about graphs and their relations, we use typed graphs and graph morphisms as presented in [22]. In this approach, graph technology is defined based on set theory, i.e., it puts additional structure on pure sets in a systematic way to cover link structures and, therefore, lifts the level of abstraction. Hence, we prefer graph technology over pure set theory as formal definition approach for models.

A graph primarily consists of a set of nodes and a set of edges interrelating nodes. Graphs may be mapped to each other componentwise by graph morphisms, i.e., nodes are mapped to nodes and edges are mapped to edges in a compatible way. Typing by meta models is implemented by morphisms that map *instance graphs* or *typed graphs* to their *type graph*. Nodes and edges are called *instance nodes/edges* and *type nodes/edges*, respectively. This basic notion of graphs may be extended by further features such as attributes, node type inheritance, and ordering of nodes. Throughout this paper, we show attributes for comprehensibility only, but omit them in the formalization, since they do not play an important role and would put additional obstacles to a broad understandability of the formal setting. However, model versioning can be formalized based on attributed graphs as well (see [22,23]). The key idea for formalizing attributed graphs is to consider attributes as special edges from graph nodes resp. edges to data type values. Ordering of nodes is shortly discussed in the context of graph merging only, while the consideration of node type inheritance is completely left out. (The interested reader can find more information about this topic in [22].) Multiplicities and containment relations used by EMF models can be formalized by graph constraints introduced later in Sect. 5. (For further information, please consider [52] and [6].)

Definition 1 (*Graph*) A *graph* $G = (G_N, G_E, s_G, t_G)$ consists of a set G_N of nodes, a set G_E of edges, as well as source and target functions $s_G, t_G : G_E \to G_N$.

Definition 2 (*Graph morphism*) Given two graphs G and H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a *graph morphism* $f : G \to H$, shortly *morphism*, if it has the following properties:

1. $f_N \circ s_G = s_H \circ f_E$ and
2. $f_N \circ t_G = t_H \circ f_E$.

If both f_N and f_E are injective, f is also called injective.

Definition 3 (*Typed graph, type graph and typing graph morphism*) A graph G is called *typed graph* or *instance graph*, if there exists a distinguished graph TG, called *type graph*, and a graph morphism $type_G : G \to TG$, called *typing graph morphism*.

In the following, we usually work with typed graphs and graph morphisms, but omit the term "typed" for better readability.

Example 1 (Statecharts modeled as typed graphs) Consider the statechart in Fig. 2b where states are represented as rounded rectangles and connected by directed edges (transitions). A state may contain substates, represented by nesting. 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 forms of statecharts. The meta-model for this

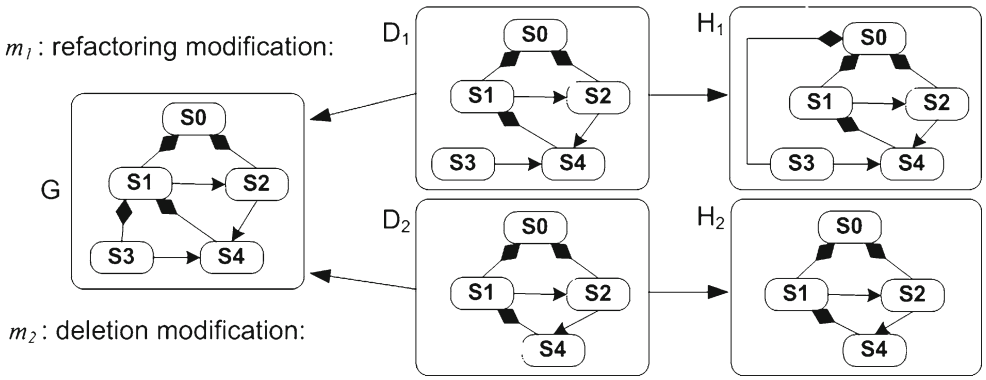


Fig. 3 Graph modifications m_1 (refactoring) and m_2 (deletion)

simplified version of statecharts is formalized as type graph shown in Fig. 2a. Here, we model hierarchical nesting of states using containment edges.

The abstract syntax of the statechart in Fig. 2b is defined by the instance graph in Fig. 2c. A node is inscribed by its identifier together with its node type. Containment edges connect a superstate with a substate. For instance, in Fig. 2c, there are containment edges from superstate S0 to its substates S1 and S2. We indicate the typing morphism by drawing some of the mappings from the instance graph to the type graph.

To be able to present meaningful versioning examples later on, we use a compact notation of the abstract syntax of statecharts, where we draw states as nodes (rounded rectangles with their node ids), mark containment edges by composition decorators on the superstate side, and depict transitions by directed arcs between state nodes. The compact notation of the statechart in Fig. 2c is shown in Fig. 2d. Note that containment and other kinds of edges do not express any ordering of nodes. If the containment of states should be ordered, then, e.g., contained elements should be connected by additional order-defining edges.

A graph modification formalizes the difference of two graphs before and after a change as a span of injective graph morphisms $G \leftarrow D \rightarrow H$ where D shows the unchanged part. This means that graph D characterizes an *intermediate graph*, where all deletion actions have been performed, but nothing has been added yet. If both graph morphisms are partial identities, this formalization suits well to model differencing where identities of model elements are preserved for each preserved element. A more general form would consider $G \leftarrow D$ to be a partial identity only allowing different identities in G and H.

Definition 4 (*Graph modification*) Given two graphs G and H, a *graph modification* $G \overset{D}{\Longrightarrow} H$ is a span of injective morphisms $G \overset{g}{\longleftarrow} D \overset{h}{\longrightarrow} H$. A sequence $G = G_0 \overset{D_1}{\Longrightarrow} G_1 \overset{D_2}{\Longrightarrow} \ldots \overset{D_n}{\Longrightarrow} G_n = H$ of graph modifications is called *graph modification sequence* and is denoted by $G \overset{*}{\Longrightarrow} H$.

Example 2 (Graph modifications) Consider the following model versioning scenario for statecharts. Two users check out the statechart shown in Fig. 2d and change it in two different ways. User 1 performs a refactoring operation. She moves state S3 up in the state hierarchy (the upper span in Fig. 3). User 2 deletes state S3 together with its adjacent transition to state S4 (the lower span in Fig. 3).

3 Detection of operation-based conflicts

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 for performing 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 shall be the source or target node of an edge to be added by another modification. Moreover, both graph modifications could delete the same graph elements. It is debatable if the common deletion of elements should still be considered as parallel independent or not. Since we consider parallel independent modifications to be performable in any order, common deletions cannot be allowed. Once modification 1 has deleted a graph element, it cannot be deleted again by modification 2. (However, we will see later in Fig. 7 that modifications with common deletions can be merged.)

Example 3 (Conflicting graph modifications) Consider once more the two graph modifications m_1 and m_2 shown in Fig. 3. Obviously, conflicts occur when user 1 tries to check in her changed graph H_1 after user 2 has checked in his changed graph H_2: state S3 has been deleted by m_2 but shall be moved

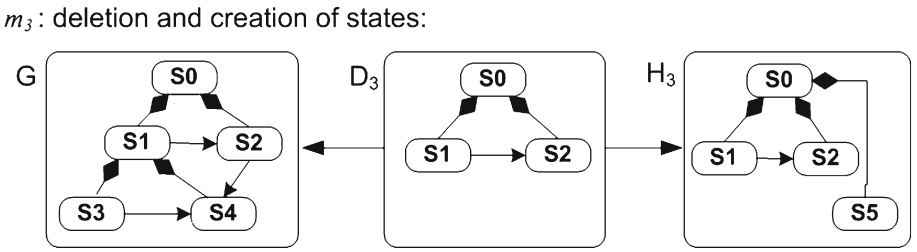


Fig. 4 Graph modification m_3 (deletion and creation)

to another container by m_1. Here, we have a *delete–insert* conflict because modification m_2 deletes node S3 that is needed by modification m_1 to insert the containment edge from state node S0 to state node S3.

Consider, in addition, a third graph modification m_3, shown in Fig. 4, that changes the same graph as m_1 and m_2 in Fig. 3. Modification m_3 deletes the substates S3 and S4 of state S1 and adds an additional state S5 as substate of S0. Here, m_2 and m_3 are in *delete–delete* conflict because both m_2 and m_3 delete the same node S3 and its adjacent edges. The additional changes defined by m_3 (the deletion of S4 and the creation of S5) do not lead to further conflicts.

The following definition formalizes these kinds of conflicts. They are operation-based conflicts, since either two deletions conflict with each other, or a deletion and an insertion are in conflict.

Definition 5 (*Operation-based conflicts of graph modifications*) Two graph modifications $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ are *in operation-based conflict* if they are in

1. delete–delete conflict, i.e., $\exists x \in (G \backslash D_1) \cap (G \backslash D_2)$ or
2. delete–insert conflict, i.e.,

$$(\exists \text{ edge } e \in H_2 \backslash D_2 \text{ with } s(e) \in D_2 \cap (G \backslash D_1) \text{ or } t(e) \in D_2 \cap (G \backslash D_1)) \text{or}$$
$$(\exists \text{ edge } e \in H_1 \backslash D_1 \text{ with } s(e) \in D_1 \cap (G \backslash D_2) \text{ or } t(e) \in D_1 \cap (G \backslash D_2)).$$

In case of attributed graphs, delete–insert conflicts can also occur if a graph element shall be deleted and an attribute of this element shall be added or changed. In those cases, attribute edges cannot be added. Conflicts with order changes, e.g., moving a node up by one modification and deleting it by another, would also lead to delete–insert conflicts.

Note that graph modifications $m_i = (G \overset{D_i}{\Longrightarrow} H_i)$, $i = 1, 2$ are the formal setting for a three-way merge where base version G is given together with two changes m_1 and m_2. It can be easily extended to $i > 2$ by comparing modifications pairwise.

4 Semi-automatic resolution of operation-based conflicts

In the following, we present a tentative merge construction for graph modifications that can always be performed, even in the presence of conflicts. As stated before, delete–delete conflicts are not real conflicts and can easily be resolved by deletion. Delete–insert conflicts, however, are not easily resolved. We propose the following procedure to deal with delete–insert conflicts. First, apply the tentative merge construction given in Definition 8 that solves delete–insert conflicts by giving priority to insertion. Deletion operations are performed as long as they do not collide with insertions. Thereafter, this tentative merge result is critically investigated concerning missing deletions. In addition, dealing with state-based conflicts is treated in the next section.

The tentative merge construction is performed stepwise: At first, all deletion actions are merged by computing the intersection of intermediate graphs yielding graph D. In case of delete–insert conflicts, D is too small and has to be extended again (to $\overline{D}$), i.e., those node deletion actions that conflict with edge insertions are taken back and the intermediate graphs of the original modifications are extended such that all insertions can now take place. Thereafter, insertions are merged. This construction is further detailed below.

This tentative merge construction does not always lead to desired results, since the standard resolution of delete–insert conflicts is not always adequate. In the second part of conflict resolution, we need to identify non-performed deletions or even further, we have to identify not or just partially performed operations that shall be resolved differently. Possible solutions are to take back the (potentially partial) execution of an operation, to complete a partial execution, to perform a different operation or a combination of those. A more detailed discussion of these resolution strategies can be found in Sect. 6.

To understand the tentative merge construction in Definition 8, we have to clarify basic operations on graphs. These are the union and intersection of graphs as well as a complement construction.

Definition 6 (*Intersection and union of graphs*) Given two graphs G and H that are subgraphs of C.

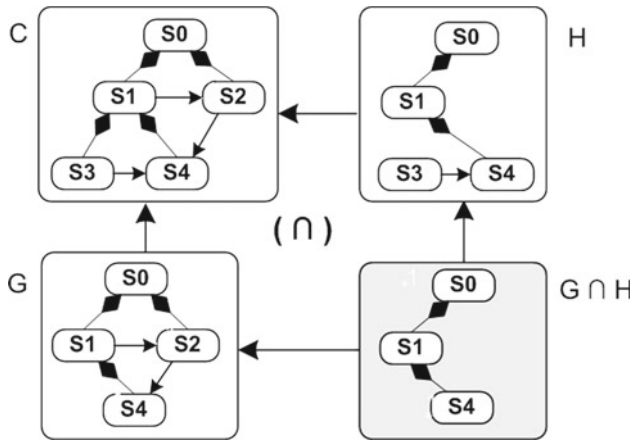


Fig. 5 Intersection $G \cap H$ of G and H over graph C

1. The intersection $G \cap H$ is defined componentwise by $(G \cap H)_N = G_N \cap H_N$ and $(G \cap H)_E = G_E \cap H_E$.
2. The union $G \cup H$ is defined componentwise by $(G \cup H)_N = G_N \cup H_N$ and $(G \cup H)_E = G_E \cup H_E$.

Example 4 (Intersection) In Fig. 5, we depict the subgraph relations from G to C and from H to C as inclusion morphisms $G \to C$ and $H \to C$, where nodes and edges are mapped identically. The intersection $G \cap H$ is the graph that contains all those nodes and edges that are present in both G and H. Obviously, the intersection graph $G \cap H$ is a subgraph of both G and H, i.e., inclusion morphisms $(G \cap H) \to G$ and $(G \cap H) \to H$ exist as shown in Fig. 5.

Example 5 (Union) In Fig. 6, we have C as a graph with both G and H as subgraphs (with inclusion morphisms $G \to C$ and $H \to C$). We can construct their intersection graph $I = G \cap H$ as in Fig. 4 and get the outer diagram in Fig. 6 as intersection diagram with inclusion morphisms $I \to G$ and $I \to H$. The union $G \cup H$ is the graph that contains the intersection graph I, and all additional elements from both $G - I$ and $H - I$. Obviously, both G and H are subgraphs also of the union graph $G \cup H$, i.e., inclusion morphisms $G \to (G \cup H)$ and $H \to (G \cup H)$ exist as shown in Fig. 6. Note that graph C in Fig. 6 is *not* the union of G and H, because in C there are additional elements like state S5 that are neither in G nor in H.

Definition 7 (*Complement graph*) Given graphs G, H, and K where H is subgraph of G and K is subgraph of H such that $\forall n \in H_N - K_N$ the following holds: $\forall e \in G_E$: $s_G(e) = n$ or $t_G(e) = n$ implies $e \in H_E - K_E$ (complement condition). The complement graph $C = (G - H) \cup K$ is defined componentwise by $C_N = (G_N - H_N) \cup K_N$ and $C_E = (G_E - H_E) \cup K_E$.

Example 6 (Complement graph) Given graphs $K \to H \to G$ as in Fig. 7.

The complement condition is satisfied since all edges in G that are adjacent to S2 (the only node in $H_N - K_N$) are also in $H_E - K_E$. Hence, we can construct the complement graph C by copying first K to C and then adding all those elements that are in $G - H$ (in our example, these elements are node S3 and its adjacent edges). Note that the complement construction yields a *union* diagram because G is the union of H and C over their common intersection graph K.

Let us consider the diagram in Fig. 8, where the complement condition is violated (there is a node S3 in $H_N - K_N$ that is also in G_N, but its adjacent edges are only in G_E and not in H_E). When trying to construct the complement graph, C should contain the edges where S3 is source or target node since these edges are not in H and not in K. But to be a valid graph, C must also contain node S3 then. Now we have the

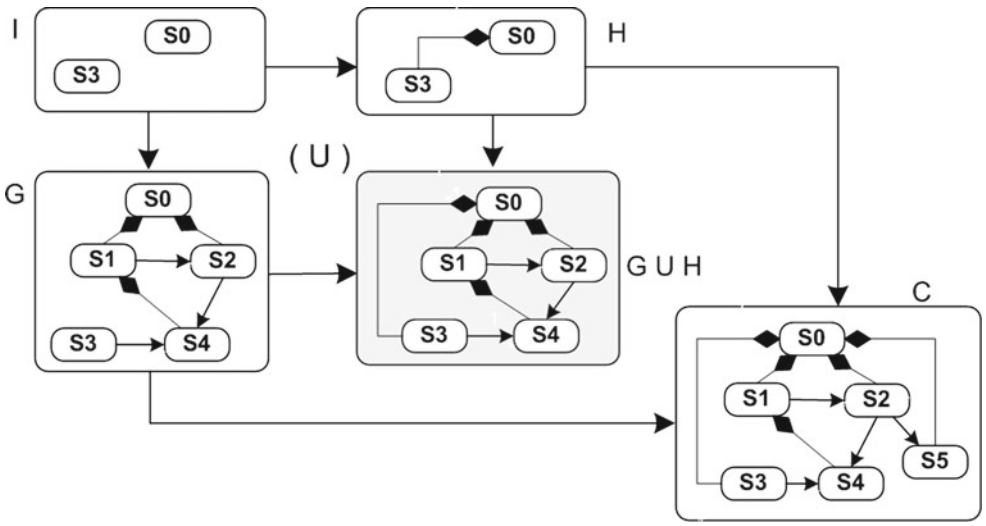


Fig. 6 Union $G \cup H$ of G and H over intersection I

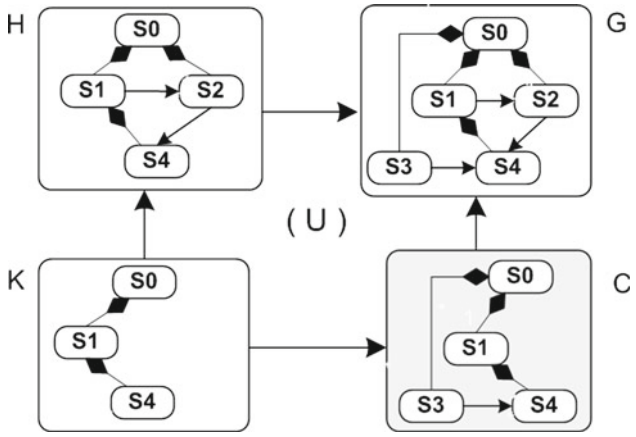


Fig. 7 Complement graph $C = (G - H) \cup K$

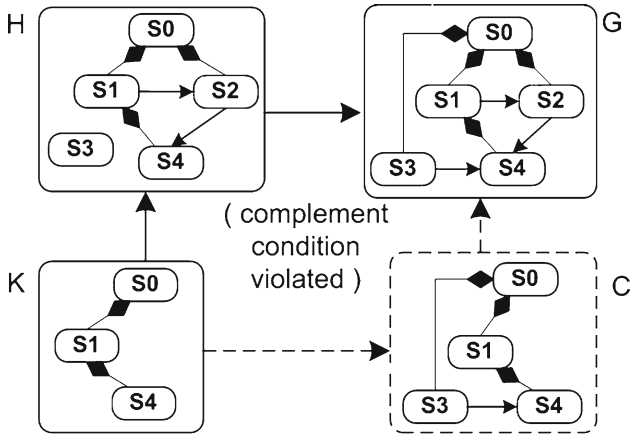


Fig. 8 No complement construction possible

situation that S3 is element of both H and C but not of K. Hence, we do not get a valid union diagram since K must be the intersection of H and C over G.

Definition 8 (*Tentative merging of two graph modifications*) Given two graph modifications $G \leftarrow D_1 \rightarrow H_1$ and $G \leftarrow D_2 \rightarrow H_2$. We construct their *tentatively merged graph modification* $G \leftarrow \overline{D} \rightarrow H$ in seven steps, leading to the following *tentative merge construction* diagram:

$$
\begin{array}{ccccc}
G & \longleftarrow & D_1 & \xrightarrow{id} & D_1 & \longrightarrow & H_1 \\
\uparrow & (\cap) & \uparrow & (=) & \uparrow & (\cup) & \uparrow \\
D_2 & \longleftarrow & D & \longrightarrow & \overline{D_1} & \longrightarrow & X_1 \\
{\scriptstyle id}\downarrow & (=) & \downarrow & (\cup) & \downarrow & (\cup) & \downarrow \\
D_2 & \longleftarrow & \overline{D_2} & \longrightarrow & \overline{D} & \longrightarrow & \overline{X_1} \\
\downarrow & (\cup) & \downarrow & (\cup) & \downarrow & (\cup) & \downarrow \\
H_2 & \longleftarrow & X_2 & \longrightarrow & \overline{X_2} & \longrightarrow & H
\end{array}
$$

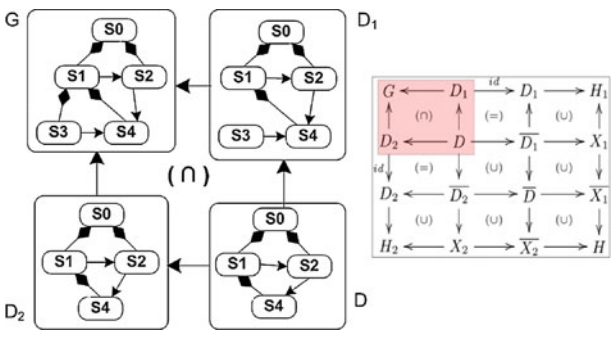
1. Construct D as intersection of D_1 and D_2 in G.
2. Extend D by nodes in D_1 that shall be deleted by modification m_2 but are needed by modification m_1. The extended graph is $\overline{D_1}$.
3. Extend D by nodes in D_2 that shall be deleted by modification m_1 but are needed by modification m_2. The extended graph is $\overline{D_2}$.
4. Unify extended graphs $\overline{D_1}$ and $\overline{D_2}$ to graph $\overline{D}$. Make sure that graphs $\overline{D_1}$ and $\overline{D_2}$ overlap in D exactly. Common supergraph is G.
5. Construct the complements $X_i = H_i - D_i \cup \overline{D_i}$ for $i = (1, 2)$. Since graphs $\overline{D_i}$ contain those nodes needed to perform modification m_i by construction, the complement condition is satisfied.
6. Unify X_i and $\overline{D}$ to $\overline{X_i}$ for $i = (1, 2)$. Make sure that graphs X_i and $\overline{D}$ overlap in $\overline{D_i}$ exactly.
7. Unify $\overline{X_1}$ and $\overline{X_2}$ to H. Make sure that $\overline{X_1}$ and $\overline{X_2}$ overlap in exactly $\overline{D}$.

$G \leftarrow \overline{D} \rightarrow H$ forms the tentatively merged graph modification.

Note that all graph morphisms in the diagram above can be considered to be inclusions. In [23], this merge construction is defined based on category theory and shown to have the intended semantics.

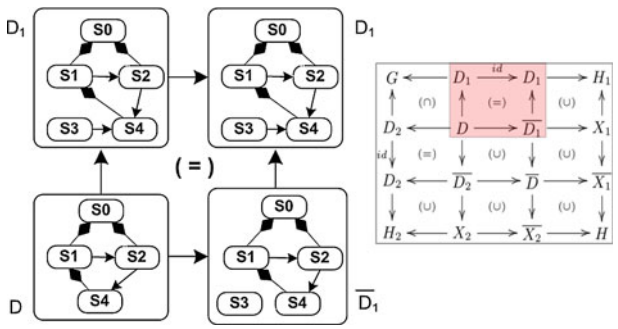
Example 7 (Tentative merge construction) We construct the tentatively merged graph modification for graph modifications $m_1 = G \leftarrow D_1 \rightarrow H_1$ and $m_2 = G \leftarrow D_2 \rightarrow H_2$ in Fig. 3. We will merge the graph modifications stepwise according to Definition 8 and highlight in a thumbnail view the current part of the construction diagram.

1. *Construct D as intersection of D_1 and D_2 in G.*
   Intersection graph D contains only those elements that are present both in D_1 and in D_2, i.e., that are preserved by both modifications. Hence, S3 and its adjacent edges are not in D.

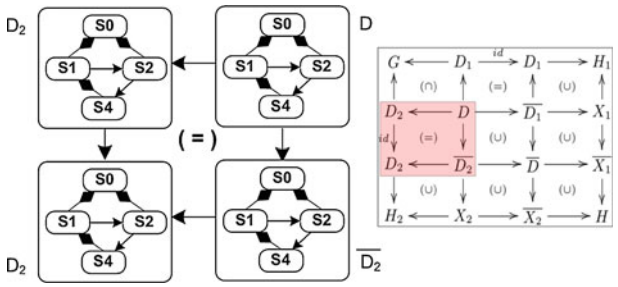


2. *Extend D by nodes in D_1 that shall be deleted by modification m_2 but are needed by modification m_1.*

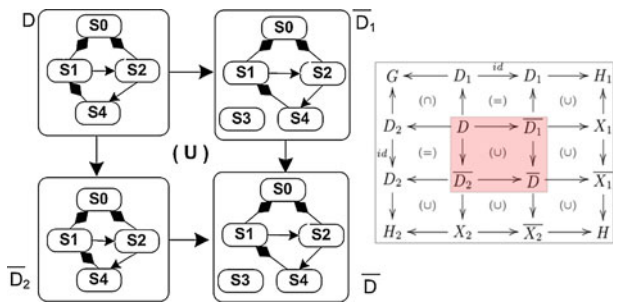
In our example, (m_2, m_1) are in delete–insert conflict. Hence, graph D is extended to graph $\overline{D_1}$ by adding node S3 since this node is deleted by m_2 but needed by m_1 to insert the new containment edge from S0 to S3.



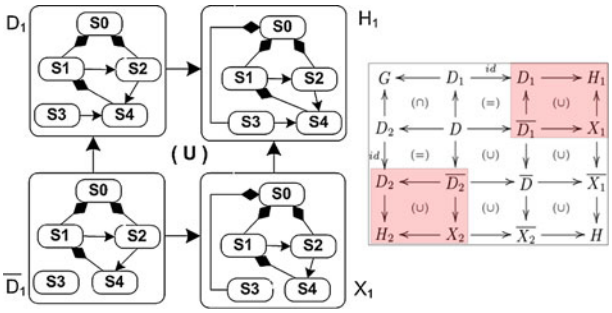
3. *Extend D by nodes in D_2 that shall be deleted by modification m_1 but are needed by modification m_2.*
This step is symmetrical to step 2. Since (m_1, m_2) are not in delete–insert conflict, $\overline{D_2} = D$.



4. *Unify extended graphs $\overline{D_1}$ and $\overline{D_2}$ to graph $\overline{D}$.*
$\overline{D}$ is constructed as union $\overline{D_1} \cup \overline{D_2}$. In addition to the common subgraph D of $\overline{D_1}$ and $\overline{D_2}$ (their intersection w.r.t. G), the union graph D contains node S3 that is in $\overline{D_1}$ but not in $\overline{D_2}$. Now, graph $\overline{D}$ contains all objects that remain after both modifications have performed their deletions *plus* those nodes that are needed for the insertion of edges by either m_1 or m_2 (node S3 in our example).

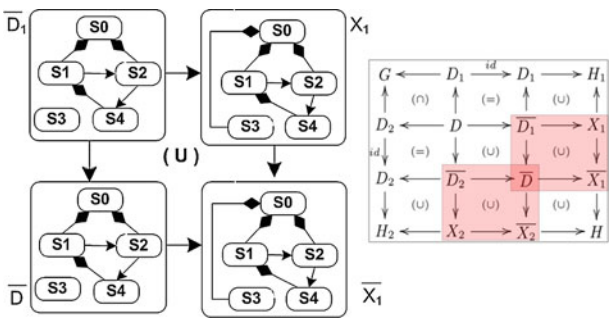


5. *Construct the complements $X_i = H_i - D_i \cup \overline{D_i}$ for $i = (1, 2)$.*
Again, the constructions of X_1 and X_2 are symmetrical. To get the union diagram, the complement graph X_1 has to contain in addition to a copy of $\overline{D_1}$ the containment edge from S0 to S3.
Since m_2 is a deletion operation, no new elements are produced, i.e., morphism $D_2 \to H_2$ is the identity. Moreover, graph $\overline{D_2}$ equals D_2 (see step 3). Hence, the complement graph X_2 (not depicted in detail) also equals D_2.



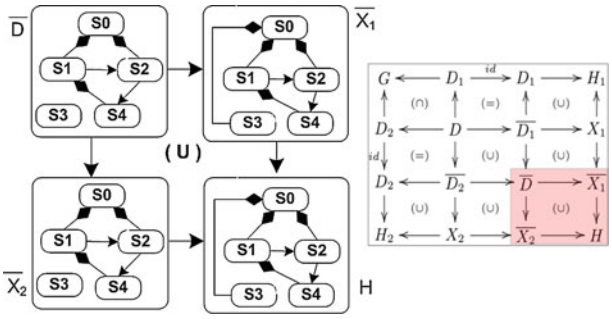
Now we have all information we need to merge the creation of elements of both modifications which is uncritical.

6. *Unify X_i and $\overline{D}$ to $\overline{X_i}$ for $i = (1, 2)$.*
These two union constructions yield the graphs $\overline{X_i} = X_i \cup \overline{D}$ that result after performing the creation of elements given by either modification on $\overline{D}$. I.e., $\overline{X_1}$ is graph $\overline{D}$ together with all elements created by m_1 (the containment edge from S0 to S3), and $\overline{X_2}$ is equal to $\overline{D}$ because m_2 does not create any elements.



7. *Unify $\overline{X_1}$ and $\overline{X_2}$ to H.*
In this last step, the creation parts of both modifications that have been performed independently of each other on $\overline{D}$ in step 6 are now merged by a union construction, leading to the tentatively merged graph H.

The tentatively merged graph modification $G \leftarrow \overline{D} \rightarrow H$ is shown in Fig. 9. It 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. Analogously, all objects deleted by either m_1 or m_2 (and not needed for edge insertion) are deleted also by the merged graph modification (e.g., the transition edge from S3 to S4).

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

Theorem 1 (Behaviour compatibility of tentative merge construction) *Given graph modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *with tentatively merged graph 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* m_1, m_2*):*

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

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* $m_2 \implies x \in G$ *preserved by* $m \implies x \in G$ *preserved by* m_1 *or* m_2
2. Deletion: $x \in G$ *deleted by* m_1 *and* $m_2 \implies x \in G$ *deleted by* $m \implies x \in G$ *deleted by* m_1 *or* m_2
3. Preservation and Deletion: $x \in G$ *preserved by* m_1 *and* $x \in G$ *deleted by* $m_2 \implies x \in G$ *preserved by* m, *if* $x \in \overline{D_1}$ $x \in G$ *deleted by* m, *if* $x \notin \overline{D_1}$ *(similar for* m_1, m_2, $\overline{D_1}$ *replaced by* m_2, m_1, $\overline{D_2}$*)*
4. Creation: $x \in H_1$ *created by* m_1 *or* $x \in H_2$ *created by* $m_2 \iff x \in H$ *created by* m

Proof See [24]. It is based on category theory.

Theorem 2 characterizes the three forms of conflict resolution that may occur.

Theorem 2 (Conflict resolution by tentative merge construction) *Given graph modifications* $m_i = G \overset{D_i}{\Longrightarrow} H_i$ $(i = 1, 2)$ *that are in conflict. The tentative 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 See [24].

If we would extend our tentative merging construction to attributed graphs, we would get the following effect: Attributed nodes that shall be deleted on the one hand and change attributes on the other hand would cause delete/insert conflicts and, therefore, would not be deleted in this merge construction. Attributes that are differently changed by both modifications would lead to attributes with two values that would cause state-based conflicts, since an attribute is not allowed to have more than one value at a particular time.

Considering ordering edges, the following conflicting merge situations could occur: Changing the order of a node on the one hand, and deleting it on the other hand, would lead to the order change while keeping this node. Moving one and the same node up and down simultaneously, would destroy a total ordering by inserting both order edges. Such a situation can be easily found by detecting state-based conflicts. Moving two subsequent nodes in a conflicting way would also destroy the total order to be resolved by repair actions.

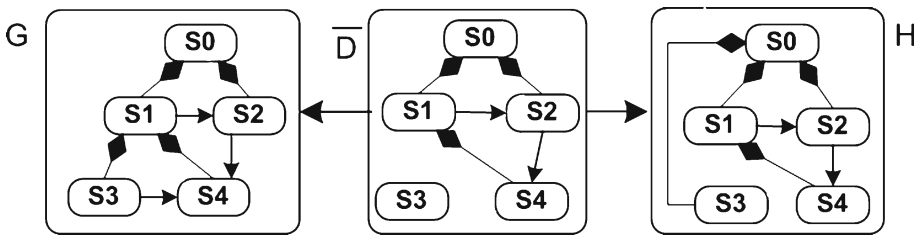


Fig. 9 Merged graph modification $G \leftarrow \overline{D} \rightarrow H$

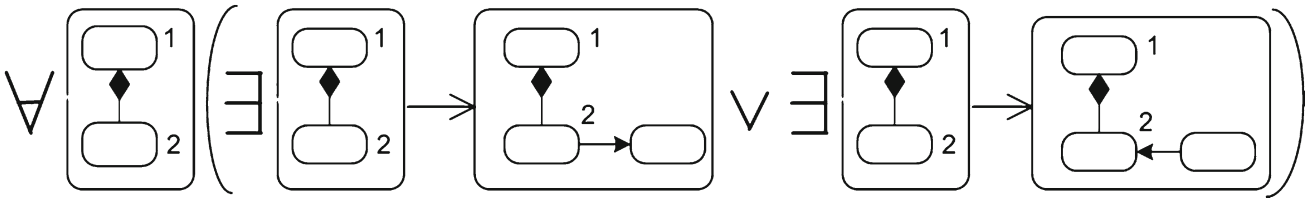


Fig. 10 Graph constraint C forbidding isolated states

The tentative merge construction can also be extended to the more general setting of more than two graph modification to be merged. In this case, two of them are merged first and then the tentative merged graph modification is merged with the next graph modification using the same construction. This procedure is continued until all original graph modifications are merged. Since merging is basically a union of all insertion and all non-conflicting deletions, the order of merging graph modifications is not significant.

5 Detection of state-based conflicts

Besides *operation-based* conflicts, we want to detect *state-based* conflicts potentially occurring in tentatively merged modification results. These conflicts occur, e.g., if a tentatively merged modification result shows some abnormality not present in the modification results before merging. Detection of state-based conflicts can be done by constraint checking. The constraints may be language-specific, i.e., potentially induced by the corresponding graph language definition. Moreover, modeling conventions can be specified by constraints and additionally checked after merging.

There exist several approaches for constraint specification in the context of modeling such as the OCL [44] for UML models and ML for colored Petri nets [27]. Since we base our approach on graphs and graph modifications, we use graph constraints [26] in the following. Since both, OCL and graph constraints, can be translated to first order logic (see [5] and [26]) they are equally powerful, and it is up to future work to show that there are natural translations between OCL and graph constraints.

Definition 9 (*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 P. Moreover, Boolean formulas of 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 an index set and c, $(c_j)_{j\in J}$ are conditions over P. Additionally, $\exists a$ abbreviates $\exists(a, \mathsf{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$, false abbreviates $\neg\,\mathsf{true}$, $\vee_{j\in J} c_j$ 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. Constraint $\exists a : \emptyset \to G$ is called simple positive constraint and abbreviated to $\exists G$. (see [26]).

Example 8 (Graph constraint) In statecharts, isolated states 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 (see Fig. 10). C is satisfied by all statecharts without isolated states.[2]

Definition 10 (*State-based conflict*) Given a merged graph modification $G \leftarrow \overline{D} \rightarrow X$ as in Definition 8, a *state-based conflict* $(C, H_1 \leftarrow \overline{D_1} \rightarrow X, H_2 \leftarrow \overline{D_2} \rightarrow X)$ consists of a constraint C and graph modifications $H_1 \leftarrow \overline{D_1} \rightarrow X$ and $H_2 \leftarrow \overline{D_2} \rightarrow X$ such that C is satisfied by graphs H_1 and H_2 but not by X.

Example 9 (State-based conflict) Consider the deletion modification m_4 in Fig. 11. Only the transition edge from S2 to S4 is deleted.

Merging this modification m_4 and the deletion modification m_2 (see Fig. 3) is unproblematic from the operational point of view: Both deletions take place at different parts of G, hence we do not have delete–delete conflicts. Moreover, neither m_4 nor m_2 does insert elements, therefore, we do not have insert–delete conflicts. Applying the tentative merging construction from Definition 8 yields X (see Fig. 12) where all deletions from m_2 and m_4 have been performed on G in any order.

We see that the tentatively merged graph X contains a forbidden situation: state S4 is isolated, i.e., it is not adjacent

[2] Note that the root container state may be isolated.

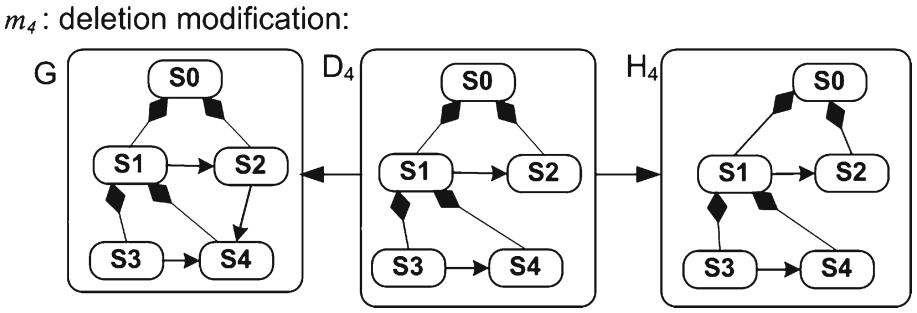


Fig. 11 Graph modification m_4: deletion of a transition

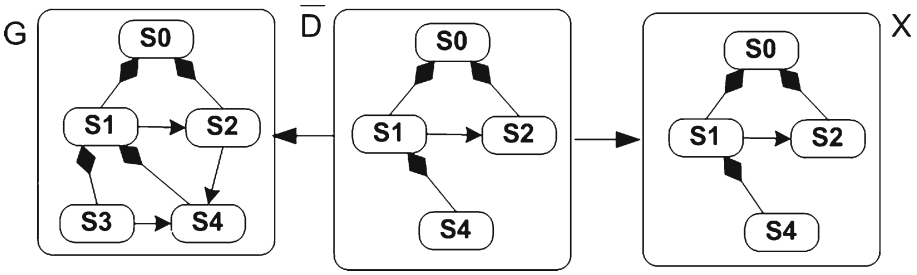


Fig. 12 Merged modification $G \leftarrow \overline{D} \rightarrow X$

to a transition anymore. Considering graph constraint C in Fig. 10, this is a state-based conflict: This constraint is satisfied for the intermediate modification results H_2 and H_4 after performing either m_2 or m_4, but it is not satisfied by X after the tentatively merged modification.

6 Resolution strategies for state-based conflicts

State-based conflicts are detected on merged graph modification, thus a natural approach to resolve them would be by executing subsequent graph modifications. We call these modifications *repair actions* in the following. In the general setting, one state-based conflict can be repaired in different ways and it is up to the user to select one of them being the most appropriate one. This situation can be compared with the detection of syntax errors in modern editors and the offering of several quick fixes to get rid of these errors.

Different specification approaches for repair actions are imaginable: Either they are explicitly defined by hand or somehow deduced from state-based conflicts. In the following, we define the general setting and give some examples for suitable repair actions.

Definition 11 (*Repair action*) Given a state-based conflict $(C, H_1 \leftarrow \overline{D_1} \rightarrow X, H_2 \leftarrow \overline{D_2} \rightarrow X)$, a repair action is a graph modification $X \leftarrow D' \rightarrow X'$ such that C is satisfied by graph X'.

Dependent on the structure of a given constraint C, we can deduce repair actions in certain cases: In case that a simple

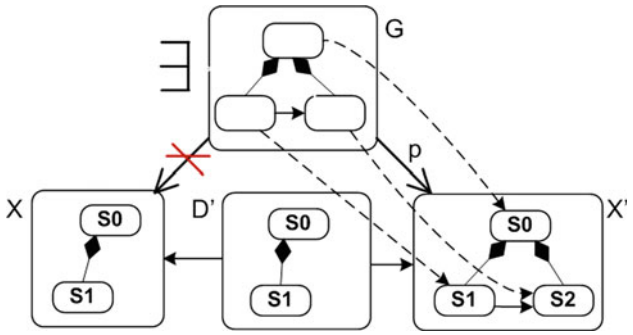


Fig. 13 Repair action for a positive constraint

positive constraint $\exists G$ is violated, e.g., subgraph G cannot be found in merge result X although it should occur. Any graph modification establishing an occurrence of G in X either by completing a partial occurrence or by creating a completely new one can be considered as repair action for this conflict. This means that different repair actions are possible in this case in general.

Example 10 (Repair action for positive constraint) Fig. 13 shows a positive constraint $\exists G$ in the top row requiring that our graph has at least two states that are connected by a transition and contained in the same container state. This constraint is violated by graph X since no occurrence $G \rightarrow X$ can be found (state S0 contains state S1 as the only substate). Our repair action $X \leftarrow D' \rightarrow X'$ establishes the missing occurrence of G in X by completing the existing partial occurrence. Alternatively, we could create the required occurrence

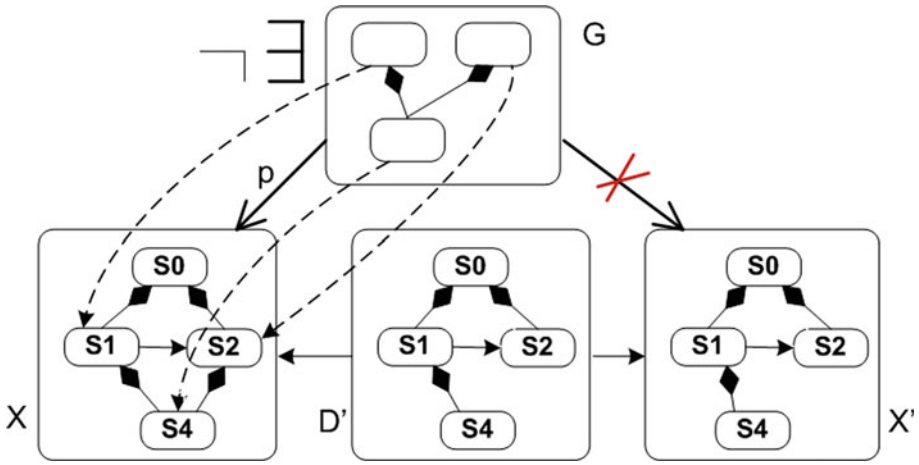


Fig. 14 Repair action for a negative constraint

completely from scratch and unite it with X. In this case, we would get a disjoint union, where not all states (not considering the root state) are contained in other states. If we prefer *rooted* statecharts, we should opt for the first alternative. We check the constraint $\exists G$ on graph X' resulting from the application of the repair action (see Fig. 13). Now we can find an occurrence of $p : G \to X'$. Hence, the positive constraint is satisfied by X'.

Analogously, in case that a simple negative constraint $\neg\exists G$ is violated, one or more occurrences of G can be found in X although such occurrences are forbidden. Repair actions have to erase all these occurrences.

Example 11 (Repair action for negative constraint) Figure 14 shows a negative constraint $\neg\exists G$ in the top row forbidding that any state is (directly) contained in more than one container state.[3] This constraint is violated by graph X since an occurrence $p : G \to X$ can be found (state S4 is contained in both states, S1 and S2). We perform the repair action $X \leftarrow D' \to X'$ that deletes one of the forbidden containment edges and check the constraint $\neg\exists G$ on the resulting graph X' thereafter. Now we do not find an occurrence of the forbidden structure G anymore, and the constraint is satisfied.

Next we consider simple implications, i.e., constraints of the form $\exists a : P \to Q$. If such a constraint is not satisfied in X, there are occurrences $p_i : P \to X$ for which there does not exist a $q : Q \to X$ with $q \circ a = p_i$ and $i \geq 1$. To repair this situation, each p_i has to be completed to some q as specified by a.

Example 12 (Repair action for implication constraint) Figure 15 shows an implication constraint requiring that each state different from the root state has an outgoing transition. This constraint is violated by graph X since for the occurrence $p : P \to X$ there is not a $q : Q \to X$ with $q \circ a = p$. We perform the repair action $X \leftarrow D' \to X'$ that adds an outgoing transition to state S4 and check the constraint $\exists a : P \to Q$ on the resulting graph X'. Now we find for the occurrence $p' : P \to X'$ (mapping state 1 to S1 and state 2 to S2) a morphism $q' : Q \to X$ with $q' \circ a = p'$. In addition, all other occurrences of P in X' have to be checked, too. It turns out that the constraint is true for all possible occurrences in X'.

We see that repair actions can be deduced from state-based conflicts, however, an exhaustive discussion of all cases is beyond this article.

7 From graph versioning to EMF model versioning

Before we proceed with presenting the implementation details of our model versioning system AMOR[4] [9], we first discuss the commonalities and differences between graphs and EMF models, i.e., models defined within the EMF [13]. First of all, each EMF model has to conform to its respective metamodel (the counterpart of type graphs for models) which is defined with the metamodeling language *Ecore*, a Java-based implementation of the MOF standard [43] provided by the Object Management Group (OMG). The core language elements of Ecore are shown in Fig. 16a in terms of a UML class diagram. Please note that we refrain from presenting all language elements and features, but concentrate on those that are of paramount importance in the context of model versioning.

Ecore EMF's metamodeling language Ecore allows to model classes, i.e., the modeling concept corresponding to type

[3] A similar containment constraint has to be valid for all objects in EMF models.

[4] http://modelversioning.org.

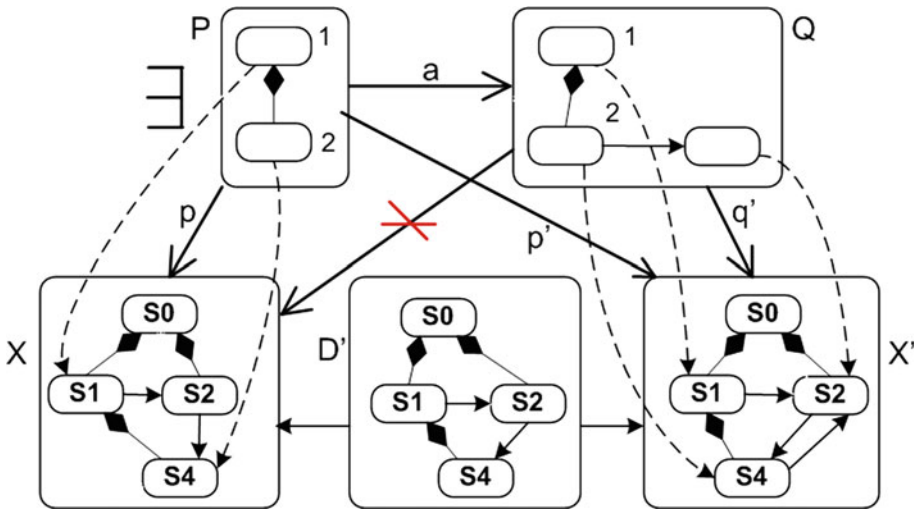


Fig. 15 Repair action for a simple implication constraint

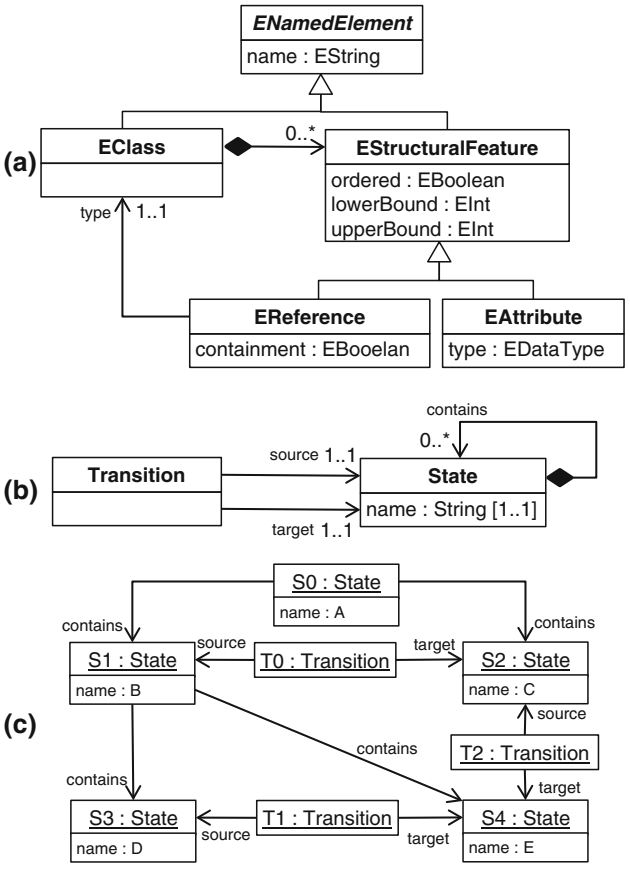


Fig. 16 EMF-based models: **a** Ecore, **b** Metamodel, and **c** Model

nodes in type graphs. Classes may contain an arbitrary number of structural features which are divided into two distinct subsets, namely references and attributes. Furthermore, for structural features, upper and lower multiplicities have to be defined.[5] Structural features having a upper multiplicity greater than 1, may be defined as *ordered*, i.e., an index is assigned to each value. Attributes as well as references must be typed. For attributes, primitive data types such as String, Boolean, and Integer are allowed. References refer to classes for defining their types and may additionally be defined as containments. Contained elements are nested inside the container element and, therefore, the deletion of a container element results in cascaded deletions of all directly and indirectly contained elements.

Graphs versus EMF models Compared to type graphs introduced in Sect. 2, Ecore models not only contain type nodes (represented by classes) and type links (represented by references), but also attribute types for classes, corresponding to attributed type graphs. Furthermore, containment references may be defined, and additional constraints to express ordering and multiplicities are possible for features. The order of values in ordered features is realized using array lists where absolute indices are assigned to each value in the list. Consequently, if a new value is inserted into an ordered list, the indices of all subsequent values are increased by one. Besides ordered features, EMF allows specifying multiplicities of attributes and references. The specification of multiplicities in graphs is also possible, by defining additional graph constraints (see [52]). As we see later, multiplicities are of special importance for merging EMF models. The upper multiplicity for features determines if a single-valued slot (upper bound equals 1) or a collection (upper bound greater than 1) is used for storing the feature value(s) in the model instances. Single-valued slots are of course problematic when two different values for the same feature of a model element are occurring, because both cannot be directly represented in the merged version.

The corresponding Ecore-based metamodel for the statechart type graph of Fig. 2a is illustrated in Fig. 16b. The differences between the type graph and the metamodel are

[5] Please note that asterisks for upper multiplicities are represented by −1 in the abstract syntax.

threefold: (i) for all features, multiplicities have been introduced, (ii) the class *State* has an attribute called name of type String, and (iii) the reference *contains* is now defined as containment.

Figure 16c shows the EMF model corresponding to the statechart graph depicted in Fig. 2c as a valid instance of the Ecore-based metamodel. For visualizing EMF-based models, we reuse the UML object diagram notation. Please note that the attribute value slots are directly contained by the objects. Furthermore, outgoing edges are also contained by the objects, and thus, are automatically deleted when the objects are deleted. In case that the target object of an edge is deleted, the target is automatically set to *undefined*. To conclude, EMF models follow the object-oriented principle of information hiding, meaning that the details of an object, namely its contained attribute values and outgoing edges, are hidden inside the object. This is different to graphs where edges are not possessed by nodes.

8 Obtaining differences between EMF models

Before considering conflicts between two concurrently modified models, we first have to address the issue of identifying and representing differences between models. Basically, there are two approaches for obtaining differences. On the one hand, they may be identified using model differencing algorithms[6] which take two versions of a model as well as their common base version as input and compute the model differences by comparing these three states. On the other hand, differences between two versions of a model may be obtained by directly recording applied changes.[7] Such approaches do not operate on the states of a model. Instead, they obtain the differences by directly recording all applied changes in the modeling environment as they are performed by the user. Both approaches have their advantages and disadvantages. In comparison to model differencing approaches, change recording is, in general, more precise and potentially enables to gather more information (e.g., the order in which the changes have been applied) than model differencing. However, these advantages come at the price of inherently strong editor-dependence because the editor used for modifying the model has to be capable of recording changes and represent them in a common format. In AMOR, we apply model differencing which is conceptually closer to the concept of graph modifications (cf. Sect. 2) than change recording because of two reasons. First, graph modifications do not represent intermediate steps within a transaction of atomic changes. In contrast, recorded change sequences may include changes that might be obsolete due to subsequent changes in the same transaction. Second, graph modifications do not comprise an order of applied changes, which is, however, usually the case with recorded changes. Just as graph modifications, model differencing approaches neither regard intermediate changes nor the order of recorded changes.

Model differencing Obtaining differences from two versions of a model is a two-phase process. First, a *match* is computed which describes the correspondences between two versions of a model. In the second phase, *differences* are obtained by a fine-grained comparison of all corresponding objects based on the afore-computed match. Consequently, the quality of the obtained differences heavily depends on the quality of the computed match. Having obtained the differences, the conflicts between two sets of differences may be identified and finally a merged version may be derived.

To provide a better overview, the relationships between the model versions, the match models, the difference models, and the conflict model are depicted in Fig. 17. In a typical model versioning scenario, there is an original model V_o that has been concurrently modified by the two modifications m_1 and m_2 leading to two revised models V_{r1} and V_{r2}. In the first step, the original version V_o is separately matched with V_{r1} and V_{r2}. From this step, two match models $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$ are obtained that describe the correspondences between V_o/V_{r1} and V_o/V_{r2}, respectively. Corresponding objects are not necessarily equal since they might have been subject to slight modifications, such as changes to attribute or reference values, between the original model and the revised model. The match model links each potentially modified object in the revised model to the corresponding original object in the original model. Thus, in the next step, based on each of the two match models, the actual differences between V_o and V_{r1} as well as V_o and V_{r2} are derived and stored as two separate difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. Difference models contain the fine-grained description of differences, i.e., attribute and reference value changes, between an original model and a revised model. Thus, they extend the match model by additional information. Finally, the two difference models are the prerequisite for the conflict detection. All identified conflicts are stored in a conflict model $C_{V_{r1},V_{r2}}$ that is the basis for deriving a merged version V_m.

In the following, we elaborate on how the match and the difference models are obtained and represented in AMOR. Furthermore, we discuss the relation of these techniques to the fundamental approach presented in Sect. 2. Please note that the metamodels introduced in the following are independent of the modeling language, i.e., the metamodel to which the matched and differenced models conform. Consequently, the correspondences and differences of *every EMF-based model* may be represented, irrespectively of its metamodel.

[6] Also referred to as *state-based versioning* [12,15,38].

[7] Also referred to as *change-* or *operation-based versioning* [15,30,36,38].

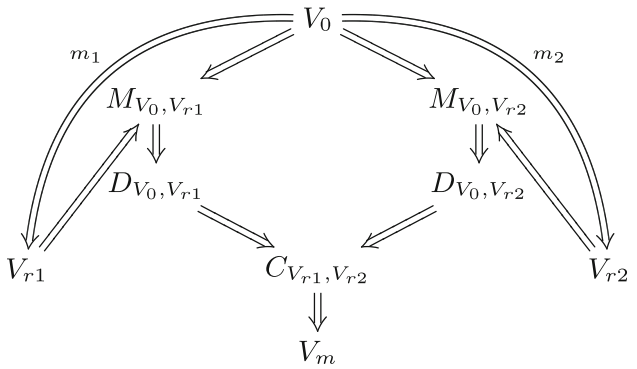


Fig. 17 Relations between Versions, Match, Difference, and Conflict Models

Match models The goal of matching two models is to produce a mapping of each object, i.e., model element, in the original version to its corresponding object in the revised version. Therefore, a match function is needed that determines whether two objects of the compared models correspond to each other. Basically, two different approaches exist for implementing such a function. First, heuristics may be employed to compute a similarity measure based on the name and other structural information between two objects. Such heuristics may strongly differ in terms of which characteristics of an object are exploited to decide whether they should be considered as a match. Of course, heuristics have an inherent amount of imprecision and are potentially very computation intense. Therefore, to overcome this drawback, the second approach avoids using heuristics by assigning a *universally unique ID* (UUID) to each object. This UUID is—once it has been assigned—never changed anymore. Consequently, it may be used to easily and efficiently match corresponding objects even if the object has been moved and/or intensely modified. However, by relying only on UUIDs, the match function misses to identify deleted and subsequently re-inserted objects which are very similar to the previously deleted ones. Even worse, some modeling environments implement moving objects by deleting and re-inserting them at a different place. Thus, a new UUID is assigned to the moved object causing the match to fail. Therefore, in AMOR we combine the advantages of both techniques by first matching all objects based on UUIDs and subsequently matching all objects that could not be matched by UUIDs using heuristics.

Match metamodel The identified correspondences are described by a match model conforming to the match metamodel depicted in Fig. 18. For each pair of matching objects, a *MatchModel* contains an instance of the class *Match* linking the corresponding object in the original version and the revised version. If an object, either in the original model or in the revised model, cannot be matched, an instance of the

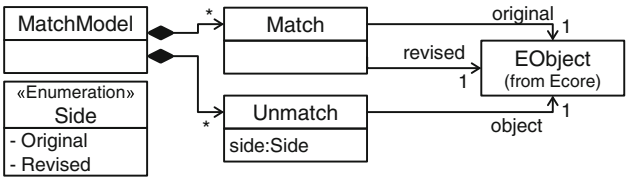


Fig. 18 Match Metamodel

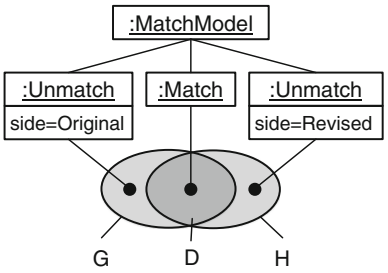


Fig. 19 Match model versus graph modification

class *Unmatch* referring to the unmatched object is created. The attribute *side* indicates whether the unmatched object resides in the original or the revised model. Hence, a match model is a kind of a *weaving model* [18].

From a conceptual point of view, the match model corresponds to the notion of graph modifications. As depicted in Fig. 19, all instances of a *Match* refer to objects of the intermediate graph D in the span $G \xleftarrow{g} D \xrightarrow{h} H$ representing a graph modification $G \overset{D}{\Longrightarrow} H$. *Unmatch* instances on the *original side* refer to deleted objects in G and *Unmatch* instances on the *revised side* to inserted objects in graph H. However, there is also a major difference due to the different representation of a model in EMF compared to the graph-based representation as specified in Definition 1. In EMF, reference values of an object are possessed by the objects themselves. Thus, they are considered as being a property of the object rather than being treated like an own entity. Consequently, in the match model only corresponding *objects* are linked by *Match* instances. In the graph-based representation however, references are represented by edges that are also included in all graphs G, D, and H. The same is true for attribute values which are not treated as own model elements in EMF but represented by own nodes in graphs. In that sense, a graph modification carries more information than a match model. This is also the reason why in EMF an additional model, i.e., the difference model, is needed to represent changes to attribute and reference values.

Example 13 (Match Model) In Fig. 20, an instance of the afore-presented match metamodel is depicted. In particular, this match model weaves the corresponding objects from the original statechart V_o represented by the graph depicted in Fig. 2d and the revised statechart V_r resulting from the modification $m2$ depicted in the lower span of Fig. 3. In this

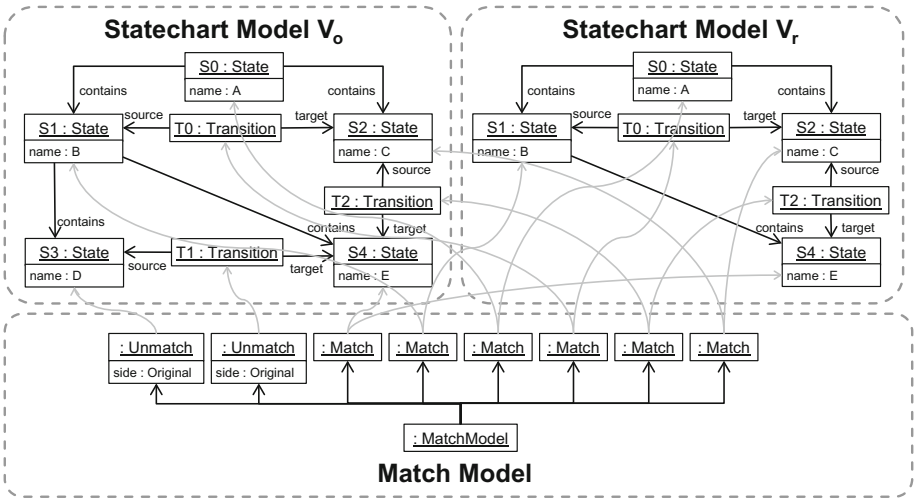


Fig. 20 Example of a match model

modification, a user deletes the state *S3* and its outgoing edge to state *S4*. According to the metamodel for statecharts depicted in Fig. 16b, an edge between state nodes is represented by the class *Transition*. Consequently, the deletion of *S3* including its outgoing edge causes the deletion of two objects, one representing the state and one representing the transition. Hence, the match model contains two *Unmatch* instances (on side *Original*) referring to each deleted object in the original model V_o. All other objects have not been deleted and no further objects have been inserted by this modification. As a result, for each remaining object, a *Match* instance is created that refers to an object in the original model V_o and to the corresponding object in the revised model V_r.

Difference models As depicted in Fig. 17, two difference models—one for each modification m_1 and m_2—are derived from the match model. Difference models comprise the information that is missing in the match model in comparison to graph modifications, i.e., changed attribute values and changed reference values. In the following, we first present a *kernel difference metamodel* that captures only the fundamental information on a model modification that is also present in graph modifications. Like graph modifications, this kernel only contains, in terms of models, additions and deletions of objects and feature values, i.e., reference and attribute values. Thus, multiplicities, ordered features, and containment features are omitted in this kernel metamodel first, but are later on considered in an extended version.

Kernel difference metamodel To represent the fundamental change types, the kernel difference model contains *FeatureChanges* as depicted in Fig. 21. Feature values in EMF models correspond to graph nodes, called *value nodes*, that are connected through edges to the possessing *object node*. According to the notion of graph modifications, nodes and edges may be inserted or deleted. In these terms, a change of a value is represented by removing the edge from the object node to the old value node and inserting a new edge to the node representing the new value. For expressing such changes in EMF models, we use two concrete subclasses of *FeatureChange* in the difference metamodel, namely *InsertFeatureValue* and *DeleteFeatureValue*. Feature changes refer to the object that has been changed (cf. reference *changedObject*), to the changed feature in the modeling language's metamodel, and to the inserted or deleted *value*. In case of a reference, this value is an object and in case of an attribute, the value is a primitive value of type String, or Boolean, etc. However, we omitted to distinguish between objects and primitive values in Fig. 21 for sake of readability. Having feature changes in difference models, every information that is present in a graph modification as presented in Sect. 2 is also expressed by a match model in combination with a difference model. However, to avoid analyzing both models when detecting conflicts, the difference model also explicitly represents inserted and deleted objects. Therefore, the metamodel contains the two classes *InsertObject* and *DeleteObject*, that are subclasses of the abstract class *ObjectChange*. Except for root objects, objects are always contained by another object through a *containment feature*. Consequently, inserting and removing an object is realized by a feature operation affecting the respective containment feature. Thus, object changes are further specified by a reference to the respective instance of a *FeatureChange*, which gives information on the inserted or deleted object through the reference *value*, the container of the inserted or removed object through the reference *changedObject* and the containment feature through which the object is or originally was contained through the reference *changedFeature*. To avoid the lengthy navigation through the referenced feature change, instances of *ObjectChange* contain a reference, called *object*, that directly refers to the inserted or deleted object. Certainly, as defined by the invariants in Fig. 21, a valid instance of *InsertObject* must

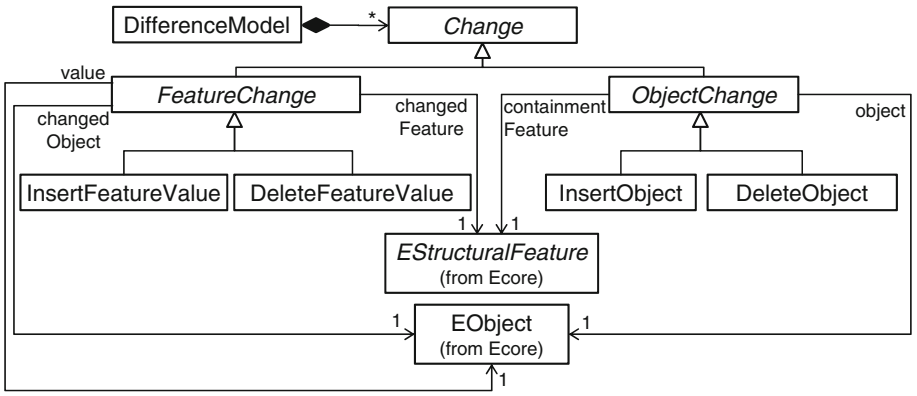


Fig. 21 Kernel difference metamodel

refer to an instance of *InsertFeatureValue* and a valid instance of *DeleteObject* must refer to an instance of *DeleteFeatureValue*, whereas the affected feature has to be a containment feature. Just like match models, also difference models are weaving models. First, a difference model refers to objects in the original model (V_o) as long as they already exist in the original model, i.e., they have not been inserted in this modification, by the references *value*, *changedObject*, and *object*. Second, it refers to the revised model (V_{r1} or V_{r2}) if the value or object has been newly introduced and, therefore, does not already exist in the original model. Third, the difference model refers to the metamodel to which the original model as well as the revised model conform by the references *changedFeature* and *containmentFeature*.

From a conceptual point of view, the algorithm for deriving a difference model from a match model is straightforward. For each *original Unmatch* instance, a *DeleteObject*, and correspondingly, for each *revised Unmatch* instance, an *InsertObject* instance is created. For deriving feature changes, each feature value of the original and revised object of each *Match* instance is compared and, given a feature value is missing on the original or the revised side, an instance of *InsertFeatureValue* or *DeleteFeatureValue* is created, respectively.

Extended difference metamodel However, when recalling the Ecore metamodel (cf. Fig. 16), it becomes obvious that the kernel difference model in Fig. 21 does not consider the complete set of Ecore modeling features and only represents the fundamental concepts that conceptually correspond to the notion of graph modifications. Several aspects, such as ordered features, containment features, and multiplicities supported by Ecore, are not covered in the kernel difference metamodel. Therefore, the kernel difference metamodel is extended to explicate all facets of changes that can be applied to Ecore-based models (cf. Fig. 22).

Firstly, in the kernel difference metamodel, the multiplicity of features is not explicitly represented. However, when merging EMF models, it makes a difference whether a feature is single-valued (*upperBound* is equal to one) or multi-valued (*upperBound* is greater than one). Changing a single-valued feature always overwrites the old value and consequently, if a single-valued feature is changed on both sides in a versioning scenario, always a conflict has to be reported. This is not the case with multi-valued features. Hence, we introduce *FeatureUpdate* that represents the change of a single-valued attribute or reference in addition to *InsertFeatureValue* and *DeleteFeatureValue* for multi-valued features.

Secondly, the Ecore metamodel allows to define *ordered* features. Ordered features pose an additional challenge when merging two versions of a model because concurrent changes of the order of feature values have to be regarded. If a feature is ordered, each element in the value list has an index. In the extended difference metamodel this is reflected by the classes *InsertOrderedFeatureValue* and *DeleteOrderedFeatureValue*. Besides inserting and deleting values from ordered feature values, users may also modify *only the order* of feature values, whereas the set of values remains the same. Such a change is realized by one instance of *DeleteOrderedFeatureValue* for detaching the object from its original index and one instance of *InsertOrderedFeatureValue* for inserting the same object at its new index again. To make changes to an order more explicit, we additionally introduce the class *FeatureOrderChange* that refers to *DeleteOrderedFeatureValue* and *InsertOrderedFeatureValue* instances realizing the order change.

Finally, we have to consider a special combination of two *FeatureChanges*, namely when two feature changes are concerned with the insertion and deletion of one and the same object in different *containers*. In such a case, we can infer that this object is *moved* from one container to another. Thus, a *Move* is a derived difference consisting of two feature changes: either one *InsertFeatureValue* and one *DeleteFeatureValue* if both containment features are multi-valued, or one *DeleteFeatureValue* and one *FeatureUpdate* if only the source containment feature is multi-valued, or one *FeatureUpdate* and one *InsertFeatureValue* if only the target containment feature is multi-valued, or two *FeatureUpdates* if

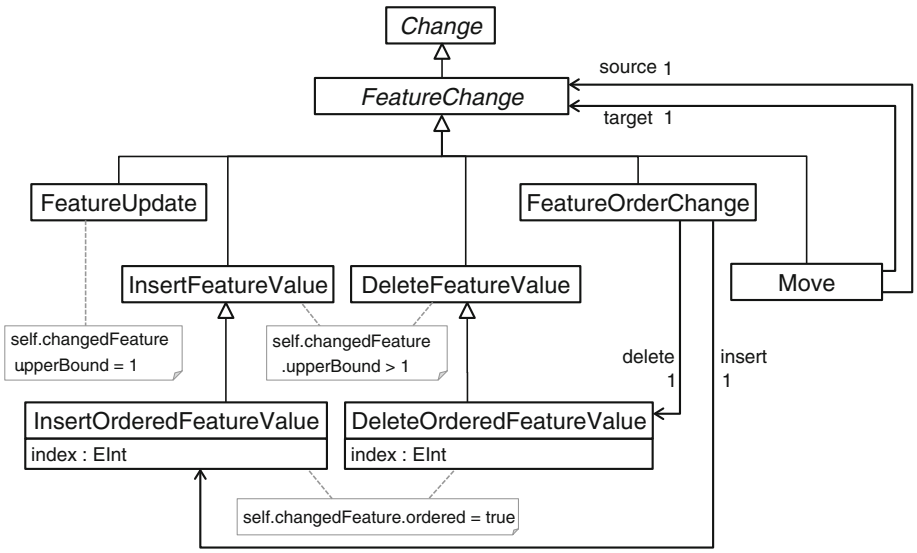


Fig. 22 Extended difference metamodel

the source and target containment features are single-valued. The old container of the moved object is indicated by the *changedObject* reference (cf. Fig. 21) in the source *FeatureChange* and the new container is indicated by *changedObject* of the target *FeatureChange*.

Technical realization The presented model differencing approach is implemented in the model versioning system AMOR. This implementation is partly based on EMF Compare [12], an extensible model differencing framework in the realm of EMF. With EMF Compare, EMF models may be matched using either heuristics or UUIDs.

However, the implementations for model matching shipped with EMF Compare have some limitations regarding completeness and flexibility. For instance, EMF Compare stops matching children of objects for which no match could be found. Consequently, an object that has been moved into a newly inserted object will not be recognized. Furthermore, it does not allow for combining UUID-based matching with heuristic matching and the used heuristics cannot be easily adapted for language-specific characteristics. The quality of the match model is however of significant importance for the subsequent model differencing and, finally, for detecting conflicts. Fortunately, EMF Compare is extensible enough to allow for replacing certain parts of it with custom implementations. Thus, we replace the default match implementation by an own implementation that first exploits UUIDs for creating an initial match model and then, for each unmatched element, tries to find additional matches based on language-specific correspondence rules that are specified using the Epsilon Comparison Language [32].

Based on this improved match model, we use EMF Compare in AMOR to derive an initial version of differences. These differences are then optimized and translated into our model-based representation as depicted in Fig. 22. The difference metamodel in EMF Compare is similar to our extended difference model, but missing some explicit information required for realizing an efficient conflict detection. Regarding conflict detection, EMF Compare also offers some capabilities to reveal basic conflicts. However, several types of conflicts are not supported. Therefore, we also created an own implementation for detecting conflicts based on two difference models conforming to our extended difference metamodel.

Example 14 (Difference Model) To exemplify the difference metamodel, a concrete instance is depicted in Fig. 23. As already mentioned, a difference model is a weaving model connecting three models: the original model V_o, the revised model V_r, and the common metamodel of V_o and V_r. Please note that Fig. 23 shows for the sake of readability only a subset of the statechart metamodel and leaves out the instance of the class *Statechart* acting as the root container of statecharts. The difference model in this example contains the differences that are derived from the match model presented in Fig. 13. To recall, in this modification, two objects, namely the state *S3* and its outgoing transition *T1*, have been deleted. Thus, the difference model contains two instances of *DeleteObject* that are further specialized by two instances of *DeleteFeatureValue*. More precisely, in the difference model, the deletion of state *S3* is represented by a *DeleteObject* instance. This object refers to the deleted object *S3* in the original model and to the instance of *DeleteFeatureValue* that describes this deletion in more detail. As this state was originally contained by the state *S1*, the instance of *DeleteFeatureValue* refers to *S1* through the reference *changedObject* as well as to the containment reference *contains*, through which *S3* was originally contained. The second instance of *DeleteObject*

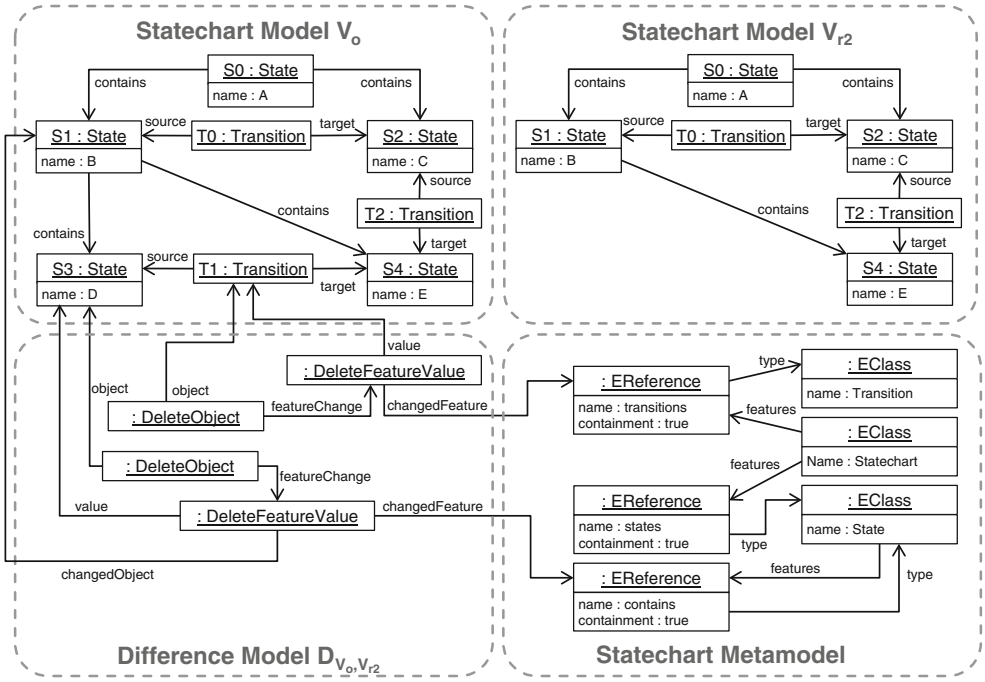


Fig. 23 Example of a difference model

indicates the deletion of the transition *T1* and also refers to the instance of *DeleteFeatureValue* that realized this deletion. Accordingly, this *DeleteFeatureValue* refers to the deleted object *T1* and the containment feature *transitions* in the statechart metamodel. Please note that we leave out the reference *changedObject* going from the *DeleteFeatureValue* for *T1* to its original container (i.e., the root object of the whole statechart), which is not depicted in Fig. 23 to avoid crowding the object diagram.

Relation to fundamental approach In this subsection, we derived the kernel difference metamodel from graph modifications and discussed how this metamodel has to be extended to cover all peculiarities of the technical space of EMF. Thereby, we aligned graph modifications and model differencing approaches and elaborated their commonalities and differences in general and in particular for EMF. As graph modifications build the basis for the subsequent steps in the conceptual versioning approach, instances of the extended difference metamodel for EMF constitute the input of the conflict detection and merging of EMF models.

9 Detection of operation-based conflicts in EMF models

Having computed the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, we may now proceed with detecting operation-based conflicts between these two difference models. Operation-based conflicts basically occur between two contradictory differences. Consequently, in the conflict detection process, we

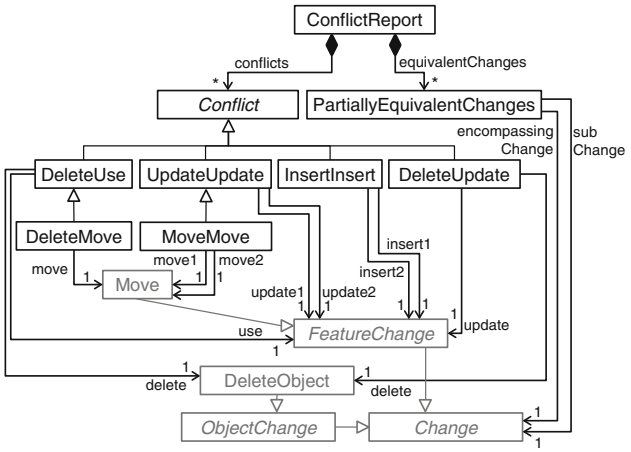


Fig. 24 Conflict metamodel

search for conflicting combinations of differences and, if a conflicting combination is at hand, the differences are marked as a conflict. In the following, we discuss the different types of operation-based conflicts potentially occurring between two difference models and illustrate them by *conflict patterns* depicted as UML Object Diagrams (cf. Figs. 25, 26, 27, 28, 29, 30, 32, 33, 35). Constraints going beyond object patterns are denoted in curly brackets using OCL, for instance, {*self.oclIsTypeOf(…)*}. Alongside the conflict patterns, we introduce a conflict metamodel to represent detected conflicts (cf. Fig. 24). Each conflict type is represented by a dedicated metaclass that is refined by additional OCL invariants stated in the afore-mentioned conflict patterns. For detecting conflicts, we may search for matches

of the conflict patterns in the difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. If a match has been found, a conflict of the respective type occurred; thus, an instance of the respective conflict metaclass is created and added to the conflict model. The created instance for describing the detected conflict is annotated in the respective conflict patterns with the stereotype $\ll$ create $\gg$. Besides occurred conflicts, the conflict metamodel (cf. Fig. 24) also describes *partially equivalent changes* that have been concurrently applied. By equivalent changes, we refer to changes that are indeed spatially overlapping, but that ultimately have the same effect and, thus, should not be marked as conflicting. This information is important for correctly creating a merged model, because in case of equivalent changes, only one of the partially equivalent changes, i.e., the encompassing change, has to be applied, and the subchange can be omitted. We discuss equivalent changes in more detail after presenting the different types of operation-based conflicts.

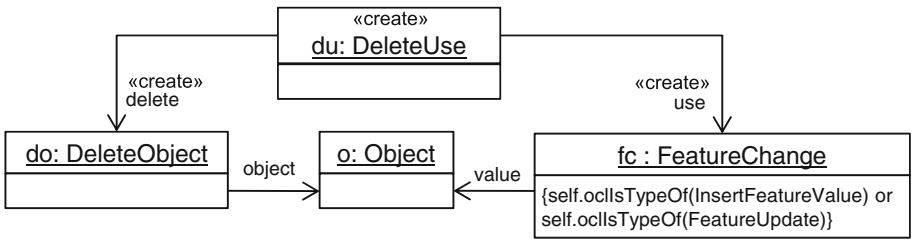
Delete–use conflict In the theory presented in Sect. 3, an operation-based conflict occurs if an edge is inserted and the source or target node of this inserted edge has been concurrently deleted. Thus, the first conflicting combination of two differences in EMF models concerns the deletion of an object and concurrently linking to exactly this object by setting a reference. We call such a conflict *delete-use* conflict because the *deleted* object is concurrently *used* as a new reference value. As defined in the conflict pattern depicted in Fig. 25, a *delete-use* conflict occurs if an object o has been deleted and the same object has been concurrently inserted as target value in a multi-valued reference or set as target value in a single-valued one. For the model-based representation of such conflicts, we introduce the class *DeleteUse* in the conflict metamodel (cf. Fig. 24). This class refers to two conflicting difference elements, namely a *DeleteObject* by the reference *delete* and a *FeatureChange* by the reference *use*. Of course, only feature changes of the type *InsertFeatureValue* or *FeatureUpdate* are valid because no conflict should be raised if a *DeleteFeatureValue* is concurrently applied.

A special kind of a *delete-use* conflict occurs when an *inserted* object *uses* a deleted object. More precisely, a user inserts an object that comprises a reference to another object that has been concurrently deleted. When objects have been inserted, the difference model contains only a difference representing the insertion; it does not contain further differences indicating the changes that have been applied to the inserted object. Thus, we require the additional conflict pattern depicted in Fig. 26 for detecting such cases. This pattern matches if an object $o1$ has been deleted (through the *DeleteObject* instance do) and a feature change fc exists that realizes the insertion of an object $o2$ having a reference to the deleted object $o1$. In the OCL invariant, we make use of the EMF-specific method called $eCrossReferences$ returning all objects that are referenced through a non-containment reference. We do not have to consider containment references in this pattern because if the deleted object has been added to a containment reference (i.e., it has been *moved* to the inserted object), the next conflict pattern called *delete-move* matches.

Delete–move conflict Another special kind of a *delete-use* conflict is a *delete-move* conflict (cf. Fig. 27) occurring if the feature change representing the *use* in a *delete-use* conflict (*fc* in Fig. 25) is part of a *Move* (cf. reference *target* in Fig. 22). As a result, moving an object and concurrently deleting the same object is indicated as *delete-move* conflict. Therefore, we introduce the class *DeleteMove* in the conflict metamodel as a subclass of *DeleteUse*.

Delete–update conflict According to Sect. 3, two graph modifications are also conflicting if a node is deleted that acts as *source* of a concurrently inserted edge. In the context of EMF models, we denote inserting, deleting, and setting feature values as an *update* of the containing object. Thus, such conflicts are denoted as *delete-update* conflicts. A *delete-update* conflict should only be raised, if the feature update is not a *DeleteFeatureValue* because in this case both changes may easily be merged without omitting the effect of one of the involved changes. Correspondingly, a *FeatureUpdate* setting

Fig. 25 Delete–use conflict



context DeleteUse
inv: self.delete.object = self.use.value **and**
(self.use.oclIsTypeOf(InsertFeatureValue) **or**
self.use.oclIsTypeOf(FeatureUpdate))

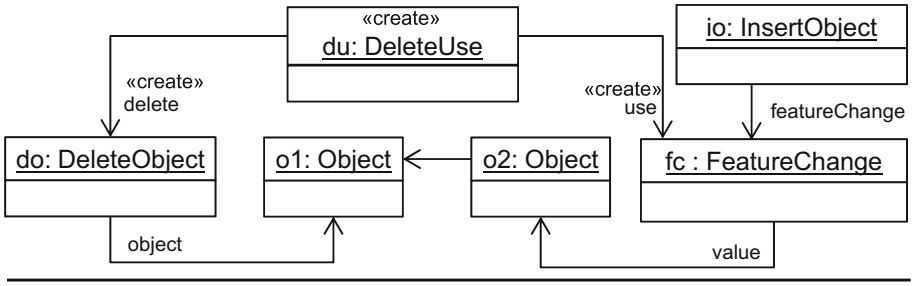


Fig. 26 Delete–use conflict: through addition

context DeleteUse
inv: InsertObject.allInstances.featureChange->includes(self.use) **and**
self.use.value.eCrossReferences->includes(self.delete.object)

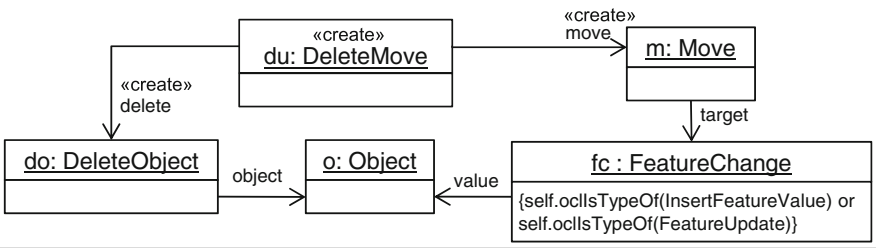


Fig. 27 Delete-move conflict

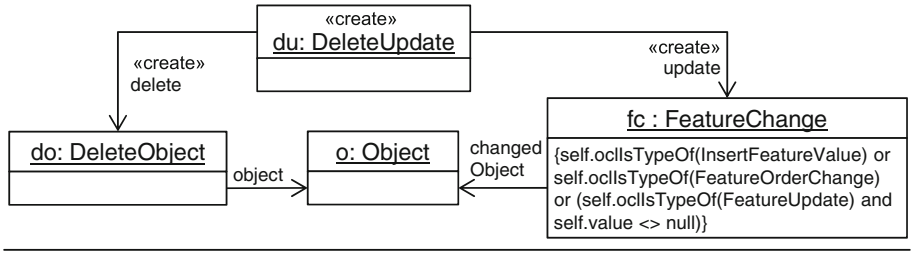
context DeleteMove
inv: self.delete.object = self.move.target.value **and**
(self.move.target.oclIsTypeOf(InsertFeatureValue) **or**
self.move.target.oclIsTypeOf(FeatureUpdate)))

a single-valued feature to *null* should also not cause a conflict when applied to an object deletion in parallel. This conceptually corresponds to the definition of operation-based conflicts in graph modifications (cf. Definition 5) and their resolution (cf. Sect. 4). According to this definition, removing an edge and concurrently removing the source node of this edge first lead to a *delete–delete* conflict that will, however, be automatically resolved by applying the deletion without user intervention. For efficiency reasons, the implementation of the conflict detection for EMF models omits raising conflicts for such scenarios in the first place. Consequently, as illustrated in the conflict pattern and the OCL invariant depicted in Fig. 28, a *delete-update* conflict occurs if an object *o* has been deleted and the same object as been updated by either an *InsertFeatureValue* or a *FeatureOrderChange*, or, in case of a single-valued feature, a *FeatureUpdate* as long as the updated value is not *null*. Please note that this pattern also raises a conflict if an object is moved to another container object that has been concurrently deleted, because the target container is updated by the *target feature change* of the *move*.

Update–update conflict As already mentioned, features in EMF models may be single-valued or multi-valued. If they are single-valued, setting a new feature value will overwrite the old one. If now a single-valued feature is concurrently changed in EMF models, obviously a conflict occurs, because the merged model may not contain both values by both users at the same time. In contrast to EMF models, we may temporarily violate the upper bound constraint in graphs: we may insert two edges to both nodes even if an upper bound of *1* is specified in the type graph. Thus, both modifications can be merged so that the merged graph ultimately contains both edges inserted by both users. If the upper bound in the type graph is now defined to be *1*, a state-based conflict is raised later (cf. Sect. 5). However, in EMF models we cannot temporarily store two values in a single-valued feature. Therefore, we rather immediately raise a conflict that is referred to as *update–update* conflict (cf. Fig. 29).

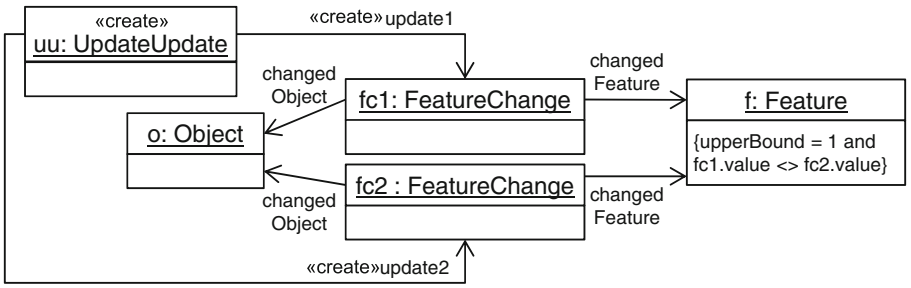
Update–update conflict: ordered features We may also encounter conflicts between concurrent changes, if the updated feature is defined to be *multi-valued* and *ordered* in the modeling language's metamodel. If the order of feature values is defined to convey a meaning, concurrent changes to such features might have contradictory effects on the order. As already mentioned (cf. Sect. 7), the order of feature values is represented in terms of absolute indices in EMF. However, based on our experiences, we argue that in most modeling languages, the meaning of a value's position in ordered features

Fig. 28 Delete-update conflict



context DeleteUpdate
inv: self.delete.object = self.update.changedObject **and**
(self.update.oclIsTypeOf(InsertFeatureValue) **or**
self.use.oclIsTypeOf(FeatureOrderChange) **or**
(self.oclIsTypeOf(FeatureUpdate) **and** self.update.value <> null))

Fig. 29 Update–update conflict



context UpdateUpdate
inv: self.update1.changedObject = self.update2.changedObject **and**
self.update1.changedFeature = self.update2.changedFeature **and**
(self.update1.changedFeature.upperBound = 1 **and**
self.update1.value <> self.update2.value)

is based on its *predecessor* and *successor*. For instance, the absolute index of messages in UML Sequence Diagrams is usually not important; the position of a message is rather characterized by its preceding and succeeding messages. Therefore, we build our conflict detection strategy for concurrent changes to ordered features upon the principle that the meaning of a value's position is constituted by its predecessor and its successor. With this strategy, we aim at raising a conflict if and only if the final order of values of a concurrently modified feature cannot be determined (because of inserting two different values at the same index) or if one change contradictorily affects the predecessor or the successor of a concurrently inserted, deleted, or reordered value. As EMF encodes the predecessor and successor of a value in terms of indices, we also have to base our conflict detection strategy on indices.

In Fig. 30, the conflict pattern formalizing our conflict detection strategy is illustrated. This conflict pattern comprises two feature changes, *fc1* and *fc2* that both modify the same object *o* at the same multi-valued ordered feature *f*. As further specified in the constraint for the object *fc2*, this conflict pattern matches if one of three particular scenarios occur: (i) Both changes are inserts at the *same* index with different values; thus, the final order of the inserted values cannot be determined. (ii) The predecessor or successor of an inserted value is concurrently deleted. (iii) The predecessor or successor of a deleted value is concurrently affected by another deletion. Please note that concurrent deletions on the same index will *not* cause a conflict. Furthermore, feature order changes are realized by a deletion and subsequent insertion of the same value at a different index; consequently, the conflict pattern in Fig. 30 also addresses conflicting feature order changes.

Example 15 (Concurrent Changes of an Ordered Feature) In Fig. 31, we show three sample scenarios for concurrent changes applied to ordered features. In the first scenario (cf. Fig. 31a), both users insert a new value to the beginning of the ordered feature. Thus, we cannot automatically determine the order and a conflict is reported (cf. case (i) in Fig. 30). In the second scenario (cf. Fig. 31b), user 2 inserts a new value *B* after *A*; this value, however, has been concurrently deleted by user 1. Thus, a conflict is reported (cf. case (ii) in Fig. 30). The third scenario (cf. Fig. 31c) illustrates the reason for

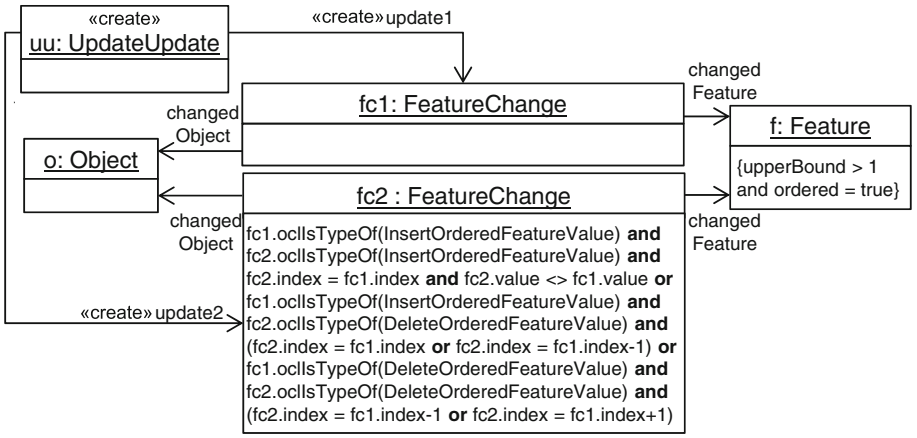


Fig. 30 Update–update conflict: ordered features

context UpdateUpdate
inv: update1.changedObject = update2.changedObject **and**
update1.changedFeature = update2.changedFeature **and**
update1.changedFeature.upperBound > 1 **and**
update1.changedFeature.ordered = true **and**
(i) update1.oclIsTypeOf(InsertOrderedFeatureValue) **and**
update2.oclIsTypeOf(InsertOrderedFeatureValue) **and**
update2.index = update1.index **and**
update2.value <> update1.value **or**
(ii) update1.oclIsTypeOf(InsertOrderedFeatureValue) **and**
update2.oclIsTypeOf(DeleteOrderedFeatureValue) **and**
(update2.index = update1.index **or** update2.index = update1.index-1) **or**
(iii) update1.oclIsTypeOf(DeleteOrderedFeatureValue) **and**
update2.oclIsTypeOf(DeleteOrderedFeatureValue) **and**
(update2.index = update1.index-1 **or** update2.index = update1.index+1)

checking the predecessor and successor index only if at least one change is a deletion (i.e., an instance of *DeleteOrderedFeatureValue*). In this scenario, both users concurrently insert values without affecting the intended predecessors and successors of the inserted values.

Move–move conflict Next, we introduce a special case of an *update–update* conflict that is related to concurrent updates of containment references of *different objects*, but using the *same object* as value. In particular, such a conflict—denoted as *move–move* conflict—occurs if the same object has been concurrently *moved* to different container objects. This is still an *update–update* conflict because *Move* is a change type consisting of two feature updates (cf. Fig. 22). However, in contrast to the common *update–update* conflicts as defined in Fig. 29, *move–move* conflicts are not caused by concurrent feature updates of the *same object* but of *different objects*. In particular, a *move–move* conflict occurs if the *same object o* has been concurrently moved to *different* container objects $c1$ and $c2$ (cf. Fig. 32). This pattern basically ensures that every object in an EMF model has at most one container. As depicted in the conflict metamodel in Fig. 24, the class *MoveMove* is a subclass of *UpdateUpdate* and additionally references two *Move* elements.

However, because of the specific restriction of EMF specifying that every EMF model must have a spanning containment tree, we also have to avoid cyclic containment relationships. Basically, a containment cycle occurs if user 1 moves an object to another container object and user 2 concurrently moves the same container object (or a parent of it) to the object user 1 moved (or a child of it). For finding such conflicts, we introduce another conflict pattern in Fig. 33. This pattern matches if one of the moved objects ($o1$) is the direct or indirect target container $c2$ of the concurrently moved object $o2$.

Example 16 (Containment Cycle) To exemplify such a scenario, we depict the original statechart and two concurrently modified versions of it V_{r1} and V_{r3} in Fig. 34. In this scenario, user 1 moves state *S3* from *S1* to *S0*. Now assume, that user 2 concurrently moves the root state *S0* to state *S3*. Obviously, a direct containment cycle between *S0* and *S3* occurs. This is the simplest case because we might also face an indirect containment cycle. Anyway, for both cases, direct and indirect containment cycles, a conflict has to be raised to avoid an erroneously merged model. For illustration purposes, we annotated the match of this conflict pattern by marking the object IDs in speech balloons at the matching model elements

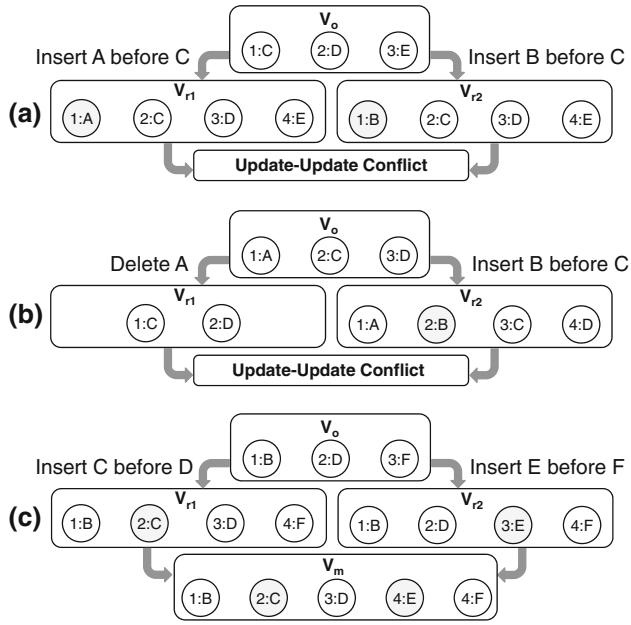


Fig. 31 Examples for concurrent changes of an ordered feature

in Fig. 34. State *S3* is a match for the conflict pattern objects *c2* and *o1* and state *S0* matches with *c1* and *o2* because the class invariant in the conflict pattern constraining *c2* specifies that $self = o1$ is fulfilled in this scenario.

Insert–insert conflict Finally, we have to regard one special case concerning the containment relationships in EMF. As already mentioned, every object must have at most one container. When considering a scenario in which one object does not have a container in the original model and both users concurrently set different containers for this object then no move–move conflict is reported. Therefore, we introduce an *insert–insert* conflict pattern for addressing such a scenarios in Fig. 35. This conflict is raised if the same object o is concurrently inserted or set as feature value of a containment reference $f1$ and $f2$ in two different objects $c1$ and $c2$. Accordingly, we also introduce the class *InsertInsert* in the conflict metamodel that references two *FeatureChanges* causing the conflict.

Completeness of the conflict patterns We developed the afore-mentioned patterns for finding operation-based conflicts, on the one hand, top-down by reviewing existing literature in the realm of conflict detection for models (e.g., [1,14,31,50,53]) and, on the other hand, bottom-up by collaboratively collecting different conflict examples from different domains in a Web-based conflict lexicon [9].

Finally, we implemented the resulting list of conflict types in AMOR and conducted several case studies in collaboration

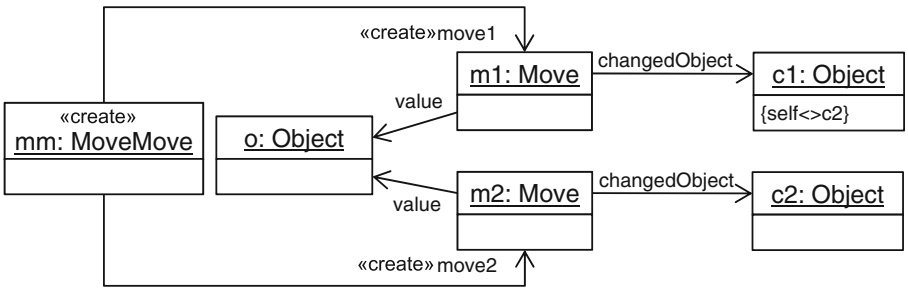


Fig. 32 Move–move conflict: non-unique container

context MoveMove
inv: self.move1.value = self.move2.value **and**
self.move1.changedObject <> self.move2.changedObject

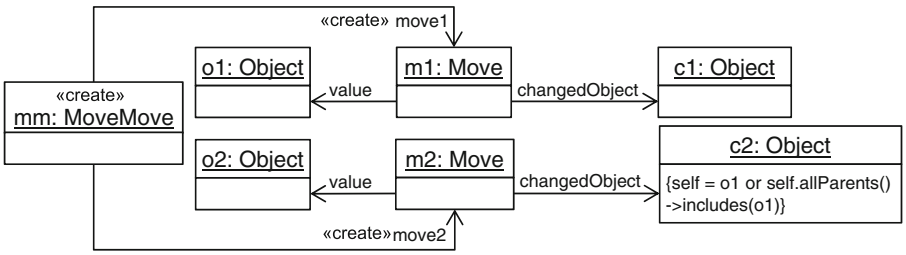


Fig. 33 Move–move conflict: containment cycles

context MoveMove
inv: self.move2.changedObject = self.move1.value **or**
self.move2.changedObject.allParents() ->includes(self.move1.value)

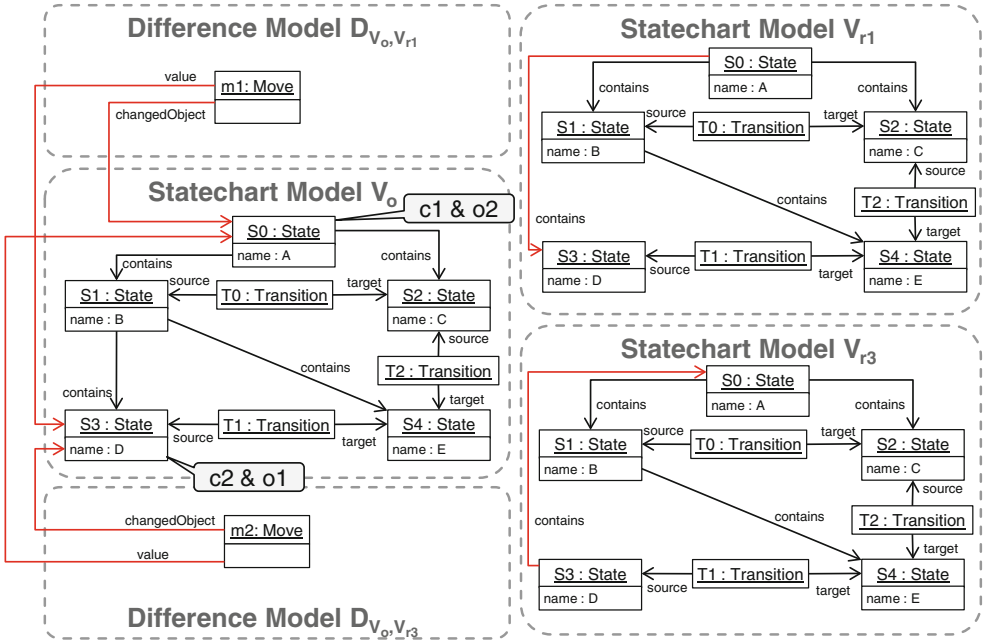
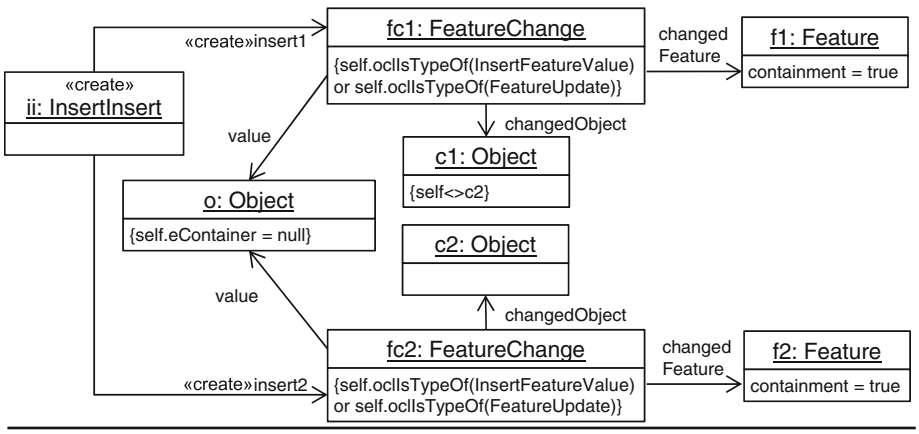


Fig. 34 Example scenario leading to a containment cycle

Fig. 35 Insert–insert conflict



context InsertInsert
inv: (self.insert1.oclIsTypeOf(InsertFeatureValue) **or**
self.insert1.oclIsTypeOf(FeatureUpdate)) **and**
(self.insert2.oclIsTypeOf(InsertFeatureValue) **or**
self.insert2.oclIsTypeOf(FeatureUpdate)) **and**
self.insert1.value = self.insert2.value **and**
self.insert1.value.eContainer = null **and**
self.insert1.changedObject <> self.insert2.changedObject **and**
self.insert1.changedFeature.containment **and**
self.insert2.changedFeature.containment

with our industry partner SparxSystems[8] (the vendor of the UML tool Enterprise Architect) to evaluate whether the list of identified conflict types covers a wide range of conflicts occurring in modeling practice. Admittedly, whether concurrent changes should be classified as conflicting often depends on how a modeling language is used, the goal of the modeling project, the phase of a project, or even on personal preferences. To this end, AMOR follows a framework approach and allows users to extend the conflict detection algorithm by adding new conflict patterns as well as changing and removing existing ones. For instance, if a very conservative versioning strategy is needed for safety-critical systems where

[8] http://sparxsystems.de.

also the concurrent modification of two different features of one object should be reported as a conflict (because the modelers should then review their modifications in combination), this is easily implementable by specifying a conflict pattern similar to the update–update conflict pattern shown in Fig. 29.

Partially equivalent changes Besides the list of conflicts, the conflict report also comprises information on (partially) equivalent changes. Changes are considered to be partially equivalent if they have (at least partly) the same effect when applied to a model. Consider for instance a scenario in which one user deletes state *S1* in the statechart depicted in Fig. 16c; due to the containment reference *contains*, also the states that are contained in *S1* are deleted (i.e., *S3* and *S4*). In parallel, user 2 deletes *S3* being a substate of *S1*. The effect of the change performed by user 2 is *partially contained* by the change applied by user 1 because both changes directly or indirectly deleted *S3*. In the previously presented conflict detection rules, we already regarded equivalent changes such that we do not report conflicts if two seemingly conflicting changes lead indeed to the same result. However, besides avoiding to report conflicts between equivalent changes, we also need to retain the information on (partially) equivalent changes for constructing a merged version (cf. Sect. 10) because, in our example, *S3* cannot be deleted anymore after its container *S1* has been deleted. Therefore, the information on equivalent changes is saved alongside the occurred conflicts in the conflict report. More precisely, if two changes are (partially) equivalent, the *smaller* change is referenced through *subChange* and the change that encompasses the *subChange* is indicated by the reference *encompassingChange* in Fig. 24. When creating the merged model, only the encompassing change is applied and the *subChange* is omitted.

Technical realization Having set up the conflict detection rules discussed above, realizing the conflict detection is largely straightforward. Generally speaking, for all change combinations of both difference models it has to be checked whether one of the afore-mentioned conflict patterns matches to indicate a conflict. However, for the sake of efficiency, we refrain from checking the complete crossproduct of all change combinations among all changes of both difference models. In contrast, both difference models are translated in a first step into an optimized view grouping all changes according to their type into potentially conflicting combinations. Secondly, all combinations are filtered out if they do not spatially affect overlapping parts of the original model. Finally, all remaining combinations are checked in detail by evaluating the previously presented patterns.

If one of the presented conflict patterns matches, a conflict description is created and added to a conflict report which is a model-based representation of all conflicting changes in two difference models. As a summary of the introduced conflict types, the complete conflict metamodel is depicted in Fig. 24. Basically, each kind of operation-based conflict is described by an instance of the specific conflict type (e.g., *DeleteUse*) referring to the two differences (depicted in gray in Fig. 24) causing the conflict. Thus, the conflict report explicitly indicates the occurred conflicts by giving for each conflict its type and the involved differences by referring to the difference models.

Example 17 (Operation-based Conflict Detection) To exemplify the conflict metamodel, we now analyze the concurrent modifications m_1 and m_2 introduced in Fig. 3. In modification m_1, user 1 *moved* state *S3* from *S1* to *S0*. In parallel, user 2 *deleted* state *S3* and its outgoing transition *T1*. Consequently, the difference model for m_1 denoted with $D_{V_o, V_{r1}}$ contains a *Move* instance. The second difference model $D_{V_o, V_{r2}}$ representing modification m_2 contains three difference elements (cf. Fig. 23), namely two *DeleteObjects*, one for state *S3* and one for transition *T1*, and a *DeleteFeatureValue* that is implied by the *DeleteObject* for *S3*. When applying all conflict detection rules above, a *delete-move* conflict is indicated between the *Move* object of *S3* and the *DeleteObject* of *S3*.

Relation to fundamental approach To cover the peculiarities of the technical space of EMF, we refined the definition of delete–insert conflicts of the fundamental approach (cf. Definition 5). This is necessary, because EMF models have to be valid graphs. More precisely, we introduced delete–use and delete–move conflicts for scenarios in which the *target of an inserted link* has been concurrently *deleted* and delete–update conflicts for scenarios in which the *source object of an inserted link* has been concurrently *deleted*. By this refinement, we have taken into account that in EMF models, links and attribute values are not first-class elements. Moreover, we presented for EMF models the operation-based conflict types update–update, move–move, and insert–insert that are covered by the fundamental approach in terms of state-based conflicts. Hence, we directly regard well-formedness rules of EMF models in the presented operation-based conflict patterns to avoid obfuscated merged models. The introduction of the update–update conflict type is necessary because (i) *single-valued* features in EMF are allowed to hold only one value at a time and (ii) some peculiarities of *ordered, multi-valued* features have to be considered. Finally, the move–move and insert–insert conflict types are introduced to reflect the fact that EMF models must form a *spanning containment tree*.

10 Construction of merged EMF models

Before we may proceed with detecting state-based conflicts, we first construct a tentatively merged model for evaluating

language-specific constraints. The goal of this merge construction is to produce a tentatively merged model, irrespectively of any occurred conflicts, to allow for the actual conflict resolution by the user. Therefore, the merge construction used in AMOR for EMF models conceptually corresponds to construction of a merged graph presented in Sect. 4. To recall, in this merge construction, *delete–delete* conflicts are resolved by performing one deletion and in case of *delete–insert* conflicts, inserts are prioritized over deletions to obtain a merged graph. For merging concurrently modified EMF models, we follow a similar strategy.

In particular, in case of *delete–use* and *delete–update* conflicts, we omit the deletions and only apply the feature updates involved in these conflicts to avoid information loss in the merged model. However, in case of *update–update* and *move–move* conflicts, we are not able to apply both conflicting changes due to the restrictions of EMF models because when applying both changes, one change would overwrite the other. For instance, having a single-valued reference that has been concurrently modified in a contradictory way, we may not express both changes in the model since EMF is only capable of persisting one value for single-valued features. However, reflecting all contradictory changes in the merged model is essential for the comprehension of all changes. The user who is responsible for manually resolving all conflicts has to understand the concurrent evolution of the model to be able to construct a consolidated version. Therefore, we overcome the restrictions of EMF models by omitting to apply both conflicting changes in case of *update–update* and *move–move* conflicts and *annotate* the merged model to support the user in understanding the evolution.

Annotating EMF models To overcome the restrictions of EMF models, we *annotate* conflicts directly in the model [10]. Unfortunately, EMF does not inherently provide a common annotation mechanism. Therefore, we ported the lightweight extension mechanism known from UML Profiles [25] to the realm of EMF models as presented in [34]. Thereby, every model may be annotated with stereotypes containing tagged values. If for instance, an *update–update* conflict appeared, a corresponding stereotype is applied to the object that was concurrently modified. This stereotype contains information on the contradictory updated values. Stereotype applications may be visualized on top of the abstract as well as the concrete syntax of a model. The annotated model acts as the basis for the actual conflict resolution by the user, who thereby may resolve all annotated conflicts directly in tentatively merged model.

Technical realization From a technical point of view, the merged model is constructed by reapplying all identified differences contained in the difference models (cf. Fig. 22) from both sides. Therefore, we implemented a dedicated model transformation engine based on the *merging framework* of EMF Compare [12] that is able to apply *FeatureChanges* and *ElementChanges* to existing EMF models. When merging all non-conflicting changes, we have to specifically treat changes to ordered features. To recall, the positions of values in ordered features are represented in EMF by indices. Thus, a change might affect the indices of all subsequent values in the ordered list. Therefore, we apply all concurrent changes to ordered features from the back to the front; for instance, a deletion of a feature value at index 7 is applied before an insertion at index 2. Besides ordered features, we also have to regard conflicting changes during the merge construction. More precisely, we follow the rules mentioned above, i.e., prioritizing feature changes over object deletions and omitting feature changes involved in *update–update* and *move–move* conflicts. As mentioned above, the tentatively merged model is finally annotated with all occurred conflicts from the conflict report using EMF Profiles to allow users to resolve them.

Example 18 (Merge Construction) To exemplify the merge construction and the annotation of conflicts, we walk through the merge and annotation process for merging the modifications m_1 and m_2 introduced in Fig. 3. In modification m_1, user 1 moved state *S3* from *S1* to *S0*. In parallel, user 2 deleted state *S3* and its outgoing transition *T1*. As elaborated in Example 17, a *delete-move* conflict is reported between the *Move* by user 1—or more precisely the *InsertFeatureValue* of *S3* from which the move has been derived—and the *DeleteElement* deleting *S3* by user 2. As stated above, in such a case, feature updates are prioritized over deletions, which is why *S3* is moved to *S0* in the merged model as depicted in Fig. 36. Additionally, a *DeleteMoveConflict* annotation is created that marks the *deleted* object as well as the source and target object of the conflicting *move*. This conflict model may be visualized in terms of annotations directly in the model as presented in [34]. Furthermore, the annotation also refers to the difference elements causing the conflict in the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. Besides handling this conflict, all non-conflicting changes are applied to the merged model. In particular, this is the deletion of the transition between *S3* and *S4*. Please note that the resulting model perfectly corresponds to the merged graph modification in Fig. 9.

To give the reader an idea, how conflicts are actually visualized in EMF-based modeling editors, a screenshot showing the afore-discussed tentatively merged model including the conflict annotation is illustrated in Fig. 37. In the modeling canvas, the state *S3* has a delete-move annotation depicted by a specific icon. Further information on the delete-move conflict is shown in the property view of the conflict annotation.

Relation to fundamental approach The merge construction for EMF implements the same strategy as introduced for

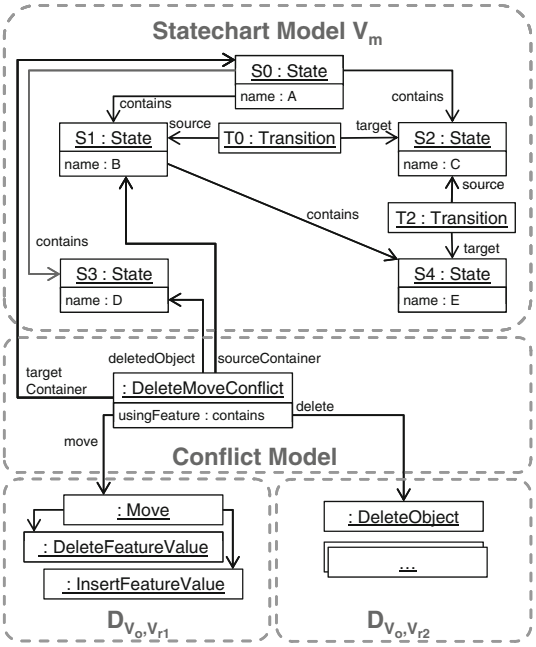


Fig. 36 Example of the tentatively merged model

the fundamental approach. More precisely, we also prioritize insertions over deletions to avoid loosing important information for resolving delete-* conflicts and, in addition, mark the conflicts explicitly by annotations. For other kinds of conflicts (i.e., update–update, move–move, insert–insert), that are reported by the fundamental approach as state-based conflicts, we do not apply both conflicting changes for building the tentatively merged model. Instead, we introduce a conflict annotation that references both changes. Thus, the user has to decide in the manual conflict resolution phase which change to prioritize. Our annotation mechanism even allows to visualize these conflict markers on top of the concrete syntax of graphical models.

11 Detection of state-based conflicts in EMF models

Having a merged model at hand, we now proceed to analyze the model to reveal state-based conflicts. Basically, we may distinguish between two kinds of state-based conflicts in the realm of EMF.

First of all, every model must conform to general well-formedness rules all EMF models have to follow regardless of their metamodels. These rules specify that every EMF model must have a spanning containment tree, i.e., every model element must be reachable from the root element following a unique path through containment references only. Thus, every model element, except the root element, must have a unique container and no cyclic containment relationships are allowed. Assuming that both modified versions V_{r1} and V_{r2} are well formed, the merged model obtained by the merge construction discussed before is also well formed because otherwise the rules for detecting *move–move*, *delete–move*, or *delete–update* conflicts would have prohibited producing a broken containment tree. Consequently, we do not have to consider containment violations anymore at this point.

Second, every model must conform to its metamodel and to potentially additional validation rules such as OCL constraints. Most of these rules coming solely from the metamodel cannot be violated in the merged model assuming that they have not been violated in each of the two concurrent versions V_{r1} and V_{r2}. Also, inserting more than one value to a single-valued feature is avoided by raising an *update–update* conflict and dangling references are prohibited by *delete–use* conflicts. However, the merged version might still violate the lower or upper bounds of multi-valued features, uniqueness constraints, and arbitrary additional constraints such as OCL constraints.

Technical realization While state-based conflicts of graph modifications are defined by graph constraints as presented in Sect. 5, we use corresponding technologies in the realm of EMF such as the EMF Validation Framework [21]. Using this framework, each EMF-based model may be validated to detect violations of constraints arising directly from the metamodel as well as those coming from additionally defined constraints. The EMF Validation Framework supports constraints expressed in OCL or Java. Whenever a violation is detected, diagnostics are returned that describe the severity of the constraint violation and provide an error message as well as the model elements involved in the respective violation.

The information on involved model elements is used to point the user to all revealed state-based conflicts. Particularly, all involved model elements are annotated using again our annotation mechanism for EMF models already introduced for annotating operation-based conflicts. This annotation provides information on which constraint is violated and which other model elements are also involved in the same violation.

Relation to fundamental approach Our approach for detecting state-based conflicts in EMF models corresponds to the fundamental approach presented in Sect. 5: we build a tentatively merged model and check the validation rules of the respective modeling language. For practical reasons, we, therefore, employ an existing technology that is specifically tailored to validating EMF models. However, the automatic resolution of state-based conflicts, as presented in Sect. 6, is currently not implemented in AMOR.

12 Related work

The contribution of this article is twofold. First, a formal foundation of model versioning concepts based on

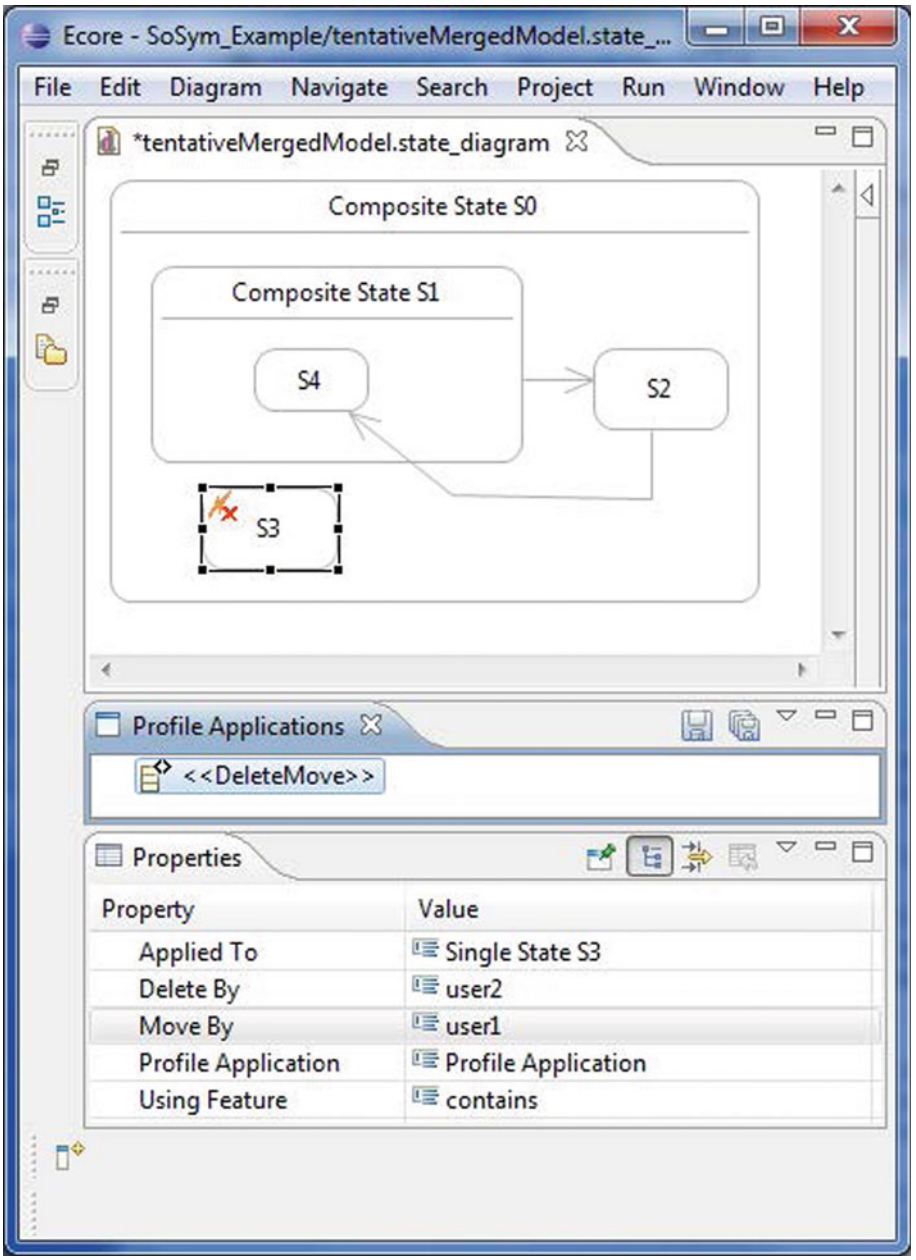


Fig. 37 Tentatively merged model in Eclipse

graph modifications is presented. Second, implementation issues and their relation to the formal foundation have been considered for the EMF technical space. Therefore, we distinguish two kinds of related work. First, we compare our work to other approaches aiming at the formalization of model versioning conflicts and then, we discuss the state-of-the-art of tool support for model versioning.

12.1 Formalization of model versioning concepts

First of all, we have to clarify that model merging differs from merging of model modifications. Model merging as presented, e.g., in [46,49] 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 are proposed that allow at least semi-automatic resolution. A survey on model versioning approaches and especially on conflict resolution strategies is given in [3].

Alanen and Porres [1] 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-update and update–update conflicts, also incorporating advanced concepts such as ordered features. However, conflicts going beyond atomic changes such as state-based conflicts remain undetected. Furthermore, delete-update and update–update conflicts have to be resolved before a merged model may be produced.

Recently, Westfechtel [53] presented a formal approach for merging EMF models. His work is based on set-theoretical conflict definitions in contrast to graph theory used in this paper. Westfechtel's approach is directly tailored to EMF models, whereas the fundamental approach presented in this paper is more generic and can be adopted to any metamodeling framework. Furthermore, in [53], only state-based conflicts arising from the well-formedness rules of EMF are regarded and no means for further language-specific constraints as discussed in Sect. 5 are provided. Our fundamental approach also encompasses automatically resolving operation-based as well as state-based conflicts which has not been considered in [53]. Consequently, in [53], all operation-based conflicts have to be interactively resolved based on an intermediate graph structure first before a merged EMF model can be created.

A category-theoretical approach formalizing model versioning is given in [48]. Similar to our approach, modifications are considered as spans of morphisms to describe a partial mapping of models. Moreover, syntactic conflicts such as adding structure to an element that has to be deleted, are identified. This kind of conflicts is very close to our delete–insert conflicts. Merging of model changes is also based on pushout constructions. In contrast to [48], we consider an automatic conflict resolution strategy that is formally defined. In addition, we consider state-based conflict detection. This has been indicated as future work in [48], where conflict detection based on user-specified operations are not mentioned at all. A category theory-based approach for model versioning in-the-large is given in [19]. However, this approach is not concerned with formalizing conflict resolution strategies.

In [33], the applied operations are identified first and grouped into parallel independent subsequences afterwards. 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 composite 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.

The approach by Blanc et al. [8,7] considers models as a sequence of construction operations. Structural constraints, i.e., constraints on model states, and methodological constraints, i.e., constraints over the model construction process itself, are formalized in consistency rules as logic formulae over a sequence of construction operations. Furthermore, the structural and methodological constraints may be defined for detecting intra-model as well as inter-model inconsistencies. In a follow-up work [41], distributed versioning based on propagating construction operations from local models to a global unified model has been formalized based on Alloy. However, only a simplified version of MOF has been considered (e.g., no multi-valued features or containment references) and there is no possibility for manual conflict resolution which we see as an integral phase of the merge process. Instead, in the work of Mougenot et al., a complete automatic merge strategy is followed. If two construction operations are in conflict, only the later one (having a newer time stamp) is integrated in the global model and the other is ignored unless it is a deletion.

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 [39] 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.

12.2 Tool support for model versioning

In the last decades, a lot of research has been conducted in the domain of software versioning which is profoundly outlined in [15,38]. Most of the approaches focus on source code versioning, others focus on two-way comparison of models [29], but there are also some dedicated approaches aiming at the versioning of models by a three-way merge. For example, Odyssey-VCS [42] 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 [12] is an Eclipse plug-in, for comparing and merging models independently of the underlying meta model. CoObRA [50] 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 [30], an Eclipse-based CASE-tool, also provides three-way merging based on edit logs. This work is continued with the development of EMF Store [31]. The Advanced Artifact

Management Systems (ADAMS) [16] has been employed for versioning software models [17] created with the UML tool ArgoUML. Cicchetti et al. [14] proposed a metamodel to describe conflict patterns used to match against a change report generated by a differentiation algorithm for detecting conflicts. In addition to research prototypes and open-source systems, current commercial modeling tools provide only some limited support for model versioning as has been evaluated in [2,4]. Most notable is the IBM Rational Software Architect (RSA), a UML modeling environment built upon EMF, providing two-way and three-way merge functionality for UML models [35].

Although all these mentioned approaches are capable of finding some kinds of operation-based conflicts, none of them allows to compute a merged version in case of conflicts are occurring. To the best of our knowledge, only two approaches, namely Mehra et al. [37] and Ohst et al. [45], exist going in this direction. In the work of Ohst et al., solely two-way merges are considered; three-way merges are only mentioned as subject to future work. Thus, only update–update conflicts for single-valued attributes arise during merging that are resolved by representing both values in the merged version similar to our approach. Update–delete conflicts are not considered, because this kind of conflict only occurs in three-way merges. The reason for this is that in two-way merges deletions cannot be detected, because no common origin model is available. In contrast to Ohst et al., Mehra et al. consider three-way merges as is done in this article. Although conflicting changes are detected by their difference algorithm, no attempt is made to indicate to the user that accepting one change may invalidate another. In our approach, we explicitly focus on detecting operation-based and state-based conflicts as well as on their automatic resolution.

13 Conclusion and future work

The main purpose of this article is to provide a fundamental basis for model versioning using graphs and graph modifications. Fundamental model versioning concepts such as model differences, conflicts, and conflict resolutions are clearly defined in a formal setting and illustrated by examples. Furthermore, we showed how these concepts can be implemented on top of existing technologies based on EMF.

Based on graph modifications as concept for model differences, operation-based and state-based conflicts are defined. Conflicts are resolved in two steps: First, a general merge construction for graph modifications with operation-based conflicts is presented that gives insertion priority over deletion in case of delete–insert conflicts. The reason for this resolution strategy is to let the merged graph modifications keep as much information as possible. We establish a precise relationship between the behavior of the given graph modifications and the merged modification concerning deletion, preservation and creation of graph items. Moreover, we discuss how different kinds of conflicts of given graph modifications are resolved by our automatic resolution strategy. It is up to additional graph modifications to perform those deletions that are preferred over insertions. These steps are intended to be performed manually by modelers.

Repair actions are provided to resolve state-based conflicts. 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 [39] as well as by Egyed et al. [47] on inconsistency checking and fixing should be considered.

Along with the clearly defined fundamental concepts, we also show how these have to be adapted to support all features and peculiarities of EMF. Furthermore, we provide deep insights into the prototypical implementation of the EMF-based model versioning system AMOR and clearly put this implementation into relation to the presented fundamental concepts. In particular, we showed how graph modifications are related to model differencing and showed which parts of the latter have been improved by considering the former. The practical relevance and the usability of AMOR and, thereby, also the corresponding fundamental concepts based on graph modifications, have been evaluated in collaboration with our industry partner SparxSystems,[9] the vendor of Enterprise Architect, in the course of user experiments. In these experiments, users had to resolve conflicts using EMF Compare as a protagonist of traditional versioning systems as well as using AMOR. These experiments showed that users appreciate to have a tentatively merged model as a basis for conflict resolution, especially when visualizing the conflicts in the concrete syntax of the models. For evaluating the performance of AMOR, we have developed a model versioning benchmark based on our previous work on establishing a collaborative conflict lexicon [11]. The benchmark consists of models that have been automatically generated using a framework for the controlled mutation of EMF-based models called Ecore Mutator.[10] For preliminary results using models comprising up to 30,000 elements and up to 500 concurrent changes, we kindly refer the interested reader to our project website.[11] The results show that the required execution time exponentially grows with an increasing number of model elements due to the state-based model differencing. An increasing number of concurrently applied changes, however, causes only a slight increase in runtime. It is worth noting that the detection of conflicts only constitutes approx. 5 % of the entire execution

[9] http://www.sparxsystems.com.

[10] http://eclipselabs.org/p/ecore-mutator.

[11] http://eclipselabs.org/p/model-versioning-benchmarks/wiki/PerformanceResultsofAMOR.

time, whereas model matching and model differencing make up approx. 37 and 58 %, respectively. In future evaluations, we plan to compare the results of AMOR with other model versioning systems.

Future work is needed to better understand the way model changes have been performed, i.e., which editing operations have been applied in which order. First ideas in this direction are described in [51] where a minimal rule is extracted from a given graph modification and that rule is compared with editing operations also specified by graph rules. In [51], we restrict our considerations on the identification of exactly one editing operation for a given graph modification. This scenario has to be extended towards an identification of a list of editing operations. Once applied operations are identified, the conflict resolution can be improved by lifting from the level of actions to the level of operations. For example, operations such as refactorings can show some variability, i.e., they can differ in their behavior depending on the context of their application. Even conflict detection and resolution on the level of single operations might be too detailed and have to be complemented with an analysis of operation sequences concerning their causality. First approaches in this direction are presented in [33] and [28]. Another interesting future research direction in the domain of model versioning concerns the graphical concrete syntax of models, which has not been addressed in this paper. When models are edited using a graphical concrete syntax, the versioning system also has to merge the concurrently changed visual representation of the models. Merging also the visual representation poses an additional challenge, because the inter-dependency between model and diagram has to be respected and the mental map [40] of the diagram has to be retained.

To conclude, model versioning is an emerging research field with new challenges concerning supportive conflict detection and resolution. A profound understanding of fundamental concepts is indispensable. Graphs and graph modifications provide a well-suited conceptual access to models and model changes on an adequate level of abstraction which helps to come up with a clear implementation of supporting tools.

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Proceedings of the International Conference on the Unififed Modeling Language (UML'03). LNCS, vol. 2863, pp. 2–17. Springer, Berlin (2003)
2. Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Why Model versioning research is needed!? An experience report. In: Proceedings of the Joint MoDSE-MCCM 2009 Workshop at MoDELS'09 (2009)
3. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. Int. J. Web Inf. Syst. **5**(3), 271–304 (2009)
4. Barrett, S., Chalin, P., Butler, G.: Model merging falls short of software engineering needs. In: Proceedings of the Workshop on Model-Driven Software Evolution (MoDSE) at MoDELS'08 (2008)
5. Beckert, B., Keller, U., Schmitt, P.H.: Translating the object constraint language into first-order predicate logic. In: Proceedings of VERIFY, Workshop at Federated Logic Conferences (FLoC) (2002)
6. Biermann, E., Ermel, C., Taentzer, G. : Precise semantics of EMF model transformations by graph transformation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28–October 3. Proceedings. Lecture Notes in Computer Science, vol. 5301, pp. 53–67. Springer, Berlin (2008)
7. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental detection of model inconsistencies based on model operations. In: Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09). LNCS, vol. 5565, pp. 32–46. Springer, Berlin (2009)
8. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), pp. 511–520. ACM, New York (2008)
9. Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kargl, H.: Adaptable model versioning in action. In: Modellierung 2010. LNI, vol. 161, GI (2010)
10. Brosch, P., Kargl, H., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G.: Conflicts as first-class entities: a UML profile for model versioning. In: Proceedings of the Models in Software Engineering Workshops at MODELS'10, Reports and Revised Selected Papers. LNCS, vol. 6627, pp. 184–193. Springer, Berlin (2011)
11. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Colex: a web-based collaborative conflict lexicon. In: Proceedings of the 1st International Workshop on Model Comparison in Practice at TOOLS'10, pp. 42–49. ACM, New York (2010)
12. Brun, C., Pierantonio, A.: Model differences in the Eclipse Modeling framework. UPGRADE Eur. J. Inform Prof. (2008)
13. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley, Reading (2003)
14. Cicchetti, A., Ruscio, D., Pierantonio, A.: Managing model conflicts in distributed development. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS'08). LNCS, vol. 5301, pp. 311–325. Springer, Berlin (2008)
15. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. **30**(2), 232–282 (1998)
16. De Lucia, A. Fasano, F., Oliveto, R., Tortora, G.: ADAMS: advanced artefact management system. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06), pp. 349–350. IEEE Computer Society, New York (2006)
17. De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: Fine-grained management of software artefacts: the ADAMS system. J. Softw. Pract. Exp. **40**(11), 1007–1034 (2010)
18. Del Fabro, M.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles (IDM'05) (2005)
19. Diskin, Z., Czarnecki, K., Antkiewicz, M.: Model-versioning-in-the-large: Algebraic foundations and the tile notation. In: Proceedings of the Workshop on Comparison and Versioning of Software Models (CVSM'09) at ICSE'09, pp. 7–12. IEEE Computer Society, New York (2009)

20. Eclipse Consortium: Eclipse Modeling Framework (EMF)—Version 2.5. http://www.eclipse.org/emf (2011)
21. Eclipse Consortium: EMF Validation Framework. http://www.eclipse.org/modeling/emf/?project=validation#validation (2011)
22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. Springer, Berlin (2006)
23. Ehrig, H., Ermel, C., Taentzer, G.: A formal resolution strategy for operation-based conflicts in model versioning using graph modifications. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'11). LNCS, vol. 6603, pp. 202–216. Springer, Berlin (2011)
24. Ehrig, H., Ermel, C., Taentzer, G.: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications (Extended Version). Technical Report 2011/1, TU Berlin (2011)
25. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. UPGRADE Eur. J. Inform. Prof. **5**(2), 5–13 (2004)
26. Habel, A., Pennemann, K.-H.: Correctness of high-level transformation systems relative to nested conditions. Math. Struct. Comput. Sci. **19**(2), 245–296 (2009)
27. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer, Berlin (2003)
28. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11). IEEE Society, New York (2011)
29. Kelter, U., Wehren, J., Niere, J.: A generic difference algorithm for UML models. In: Proceedings of Software Engineering 2005. LNI, vol. 64, pp. 105–116, GI (2005)
30. Kögel, M.: Towards software configuration management for unified models. In: Proceedings of the Workshop on Comparison and Versioning of Software Models at ICSE'08, pp. 19–24. ACM, New York (2008)
31. Kögel, M., Helming, J.: EMFStore: a model repository for EMF models. In: Proceedings of the 32nd International Conference on Software Engineering, vol. 2 (ICSE'10), pp. 307–308. ACM, New York (2010)
32. Kolovos, D.: Establishing correspondences between models with the epsilon comparison language. In: Proceedings of the 5th European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA'09). LNCS, vol. 5562, pp. 146–157. Springer, Berlin (2009)
33. Küster, J.M., Gerth, C., Engels, G.: Dependent and conflicting change operations of process models. In: Proceedings of the 5th European Conference on Model Driven Architecture—Foundations and Applications (ECMDA-FA'09). LNCS, vol. 5562, pp. 158–173. Springer, Berlin (2009)
34. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: From UML profiles to EMF profiles and beyond. In: Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11). LNCS, vol. 6705, pp. 52–67. Springer, Berlin (2011)
35. Letkeman, K.: Comparing and merging UML models in IBM Rational Software Architect: Part 3—a deeper understanding of model merging. Technical report, IBM Rational (2005)
36. Lippe, E., van Oosterom, N.: Operation-based merging. In: ACM SIGSOFT Symposium on Software Development Environment, pp. 78–87. ACM, New York (1992)
37. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05), pp. 204–213. ACM, New York (2005)
38. Mens, T.: A state-of-the-art survey on software merging. IEEE Trans. Softw. Eng. **28**(5), 449–462 (2002)
39. Mens, T., van der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06). LNCS, vol. 4199, pp. 200–214. Springer, Berlin (2006)
40. Misue, K., Eades, P., Lai, W., Sugiyama, K.: Layout adjustment and the mental map. J. Vis. Lang. Comput. **6**(2), 183–210 (1995)
41. Mougenot, A., Blanc, X., Gervais, M.-P.: D-Praxis: a peer-to-peer collaborative model editing framework. In: Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS). LNCS, vol. 3442, pp. 16–29. Springer, Berlin (2009)
42. Murta, L., Corrêa, C., Prudêncio, J.G., Werner, C.: Towards Odyssey-VCS 2: improvements over a UML-based version control system. In: Proceedings of the 2nd International Workshop on Comparison and Versioning of Software Models at ICSE'08, pp. 25–30. ACM, New York (2008)
43. Object Management Group (OMG): Meta Object Facility, Version 2.0. http://www.omg.org/spec/MOF/2.0/PDF/ (2006)
44. Object Management Group (OMG): OCL Specification—Version 2.2. http://www.omg.org/spec/OCL/ (2010)
45. Ohst, D., Welle, M., Kelter, U.: Differences between Versions of UML Diagrams. In: Proceedings of the 9th European Software Engineering Conference (ESEC'03), pp. 227–236. ACM, New York (2003)
46. Pottinger, R., Bernstein, P.A.: Merging models based on given correspondences. In: Proceedings of the International Conference on Very Large Data Bases (VLDB'03), pp. 826–873. VLDB Endowment (2003)
47. Reder, A., Egyed, A.: Model/analyzer: a tool for detecting, visualizing and fixing design errors in UML. In: Proceedings of the 25th International Conference on Automated Software Engineering (ASE'10), pp. 347–348. ACM, New York (2010)
48. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A category-theoretical approach to the formalisation of version control in MDE. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'09). LNCS, vol. 5503, pp. 64–78. Springer, Berlin (2009)
49. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency checking of conceptual models via model merging. In: Proceedings of the International Conference on Requirements Engineering (RE'07), pp. 221–230. IEEE, New York (2007)
50. Schneider, C., Zündorf, A., Niere, J.: CoObRA—a small step for development tools to collaborative environments. In: Workshop on Directions in Software Engineering Environments at ICSE'04 (2004)
51. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict detection for model versioning based on graph modifications. In: Proceedings of the International Conference on Graph Transformations (ICGT'10). LNCS, vol. 6372, pp. 171–186. Springer, Berlin (2010)
52. Taentzer, G., Rensink, A.: Ensuring structural constraints in graph-based models with type inheritance. In: Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'05). LNCS, vol. 3442, pp. 64–79. Springer, Berlin (2005)
53. Westfechtel, B.: A formal approach to three-way merging of EMF models. In: Proceedings of the Workshop on Model Comparison in Practice (IWMCP) at TOOLS'10, pp. 31–41. ACM, New York (2010)

Author Biographies

Gabriele Taentzer (e-mail: taentzer@mathematik.uni-marburg.de) is Professor for Software Engineering at the Department of Mathematics and Computer Science of the Philipps-Universität Marburg in Germany. She achieved the habilitation in Computer Science at the Technische Universität Berlin in 2003. Her research interests include model-driven software development, especially domain-specific visual languages, model transformation and model quality assurance, and graph transformation. She is steering committee member for conferences on automated software engineering (ASE), ETAPS, and graph transformation (ICGT). Furthermore, she has been program committee member of a variety of international conferences, especially on model-driven engineering such MoDELS, ECMFA, and ICMT. She has been involved in a number of research projects on model-driven software development and graph transformation.

Claudia Ermel (e-mail: claudia.ermel@tu-berlin.de) is employed at the Department of Software Technology and Theoretical Computer Science at Technische Universität Berlin (TUB), Germany since 1989. She received her doctoral degree in 2006. In her Ph.D. thesis, she developed a framework for graph transformation-based animation of visual languages. Her research interests are the application of graph transformation techniques and tools to visual language engineering, visual modeling and model transformations in model-driven development. For the last 5 years, she has been teaching the tool development of visual modeling environments in Eclipse, based on EMF, GEF and graph transformation, in student projects at TUB.

Philip Langer is postdoctoral researcher in the Business Informatics Group at the Vienna University of Technology. Before that, he was researcher at the Department of Telecooperation at the Johannes Kepler University Linz and received a Ph.D. degree in computer science from the Vienna University of Technology in 2011 for his thesis on model versioning and model transformation by demonstration. His current research is focused on model evolution, model transformations, and model execution in the context of model-driven engineering. For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/planger or contact him at email:langer@big.tuwien.ac.at.

Manuel Wimmer is working as a post-doc researcher at the Business Informatics Group of the Vienna University of Technology. His research interests comprise Web engineering and model engineering; in particular, model transformations based on formal methods, generating transformations by-example as well as applying model transformations to deal with model (co-)evolution. Currently, he is on leave working as visiting researcher at the Software Engineering Group of the University of Málaga (Spain). For further information about his research activities, please visit http://www.big.tuwien.ac.at/staff/mwimmer or contact him at email: wimmer@big.tuwien.ac.at.