A Formal Approach to Three-Way Merging of EMF Models

Bernhard Westfechtel
Applied Computer Science I, University of Bayreuth, D-95440 Bayreuth
bernhard.westfechtel@uni-bayreuth.de

ABSTRACT

Inadequate version control for models significantly impedes the application of model-driven software development. In particular, sophisticated support for merging model versions is urgently needed. In this paper, we present a formal approach to three-way merging of models in the EMF framework which may be applied to instances of arbitrary Ecore models. We specify context-free and context-sensitive rules for model merging which both detect and resolve merge conflicts. Furthermore, we present a merge algorithm which produces a valid model provided it is supplied with valid input models.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*software configuration management*

General Terms

Algorithms, Theory

Keywords

EMF models, merging

1. INTRODUCTION

Model-driven software development strives for reducing the effort of developing software by replacing conventional programming with the construction of high-level, executable models. Currently, model-driven software development constitutes a hot research topic which is addressed by more and more dedicated conferences and workshops. Furthermore, model-driven software development is starting to make its way into industrial practice.

However, model-driven software development is not a mature technology yet, and many open problems still call for adequate solutions. In particular, inadequate *version control* has been identified as a major obstacle to practical applications. Traditional version control operates on text files. It has been recognized early that *text-based tools* do not produce satisfactory results for version control of models. In particular, this applies to *comparing* and *merging versions*, the latter of which is addressed in this paper.

The shortcoming of text-based tools has motivated the development of *model-based tools*, i.e., tools which take the structure of the model versions into account. Quite a number of approaches to *model merging* have been proposed. Unfortunately, the reasoning capabilities realized in these approaches are still limited, and the model merge may produce an inconsistent result [11, 3].

In this paper, we describe an approach to model merging in the Eclipse Modeling Framework (EMF) which (1) may be applied to all models instantiated from Ecore models, (2) performs a *three-way merge* of two alternative versions with respect to a common base version, (3) is based on *formally defined merge rules*, (4) is able to handle *moves* of model elements in addition to insertions, deletions, and updates, (5) produces a *valid* (syntactically correct) *model* as result, and (6) detects and resolves both *context-free* and *context-sensitive conflicts* (contradictory changes to the same element and to different elements, respectively).

The approach serves as a *formal specification* based on which a tool for model merging may be realized. We consider such a formal specification essential because it precisely describes the problem to be solved on a high level of abstraction. To obtain a declarative and concise specification, rules for merging and conflict detection are formalized in logic. In addition, some high-level graph algorithms are given where the purely declarative logic notation is not expressive enough.

2. PRELIMINARIES

Three-way merging is used to combine two *alternative versions* derived from a common *base version* into a single *merged version.* Non-conflicting changes relative to the base version are included automatically into the merged version. *Conflicts* occur in the case of contradictory changes and are resolved either automatically or interactively.

Figure 1 illustrates alternative approaches to three-way merging. In the case of *state-based merging* [19], the states of the alternative versions a_1 and a_2 and the base version b are taken as inputs. The versions are compared pair-wise along three ways (illustrated by the edges of the triangle). A merged version m is constructed which incorporates elements from a_1 and a_2 (illustrated by the arrows ending at m) with the intent to reconcile the changes having been applied concurrently to b. In the case of differences between

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWMCP '10, July 1, 2010 Malaga, Spain
Copyright 2010 ACM 978-1-60558-960-2 ...$10.00.

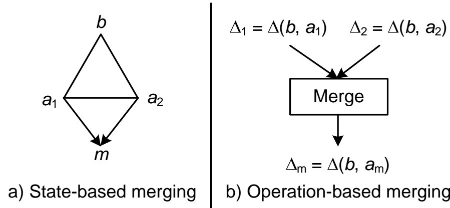

Figure 1: Three-way merging

a_1 and a_2, the base version is inspected to figure out which changes have been performed on each branch. If a change has been performed only on one branch, it is applied automatically. If (contradictory) changes have been performed on both branches, a conflict is reported.

Alternatively, *operation-based merging* [16] takes the base version b and the *deltas* Δ_1 and Δ_2 (sequences of change operations) from the base to the alternative versions as inputs. The deltas are combined in an order-preserving way to produce a *merged delta* Δ_m which is used to create m from b. The merge process has to eliminate *duplicate operations* and has to detect conflicts. Here, a *conflict* between two operations $op_1 \in \Delta_1$ and $op_2 \in \Delta_2$ occurs in some state s reached during the merge if either one of the operations *invalidates* the other one (Condition 1) or both operations are applicable in sequence, but they *do not commute* (Condition 2):

$$op_2(op_1(s)) = \bot \vee op_1(op_2(s)) = \bot \quad (1)$$

$$op_2(op_1(s)) \neq \bot \wedge op_1(op_2(s)) \neq \bot \wedge$$
$$op_2(op_1(s)) \neq op_1(op_2(s)) \quad (2)$$

State- and operation-based merging are alternative ways to solve the same problem, namely to reconcile concurrent changes having been applied to a common base version. In state-based merging, the deltas are reconstructed implicitly by comparing the states. In our approach, we have decided to apply *state-based merging* for two reasons:

First, operation-based merging suffers from a *missing comparison*. It operates on two rather than on three deltas. As a consequence, duplicate operations may go unnoticed. E.g., when the "same" object is created on both branches, it may be assigned different object identifiers. Thus, it is not recognized that operations actually apply to the "same" object.

Second, operation-based merging also suffers from *missing state information*. When constructing the merged delta step by step, we have to decide for the next operation to be processed whether it stands in conflict with some future operation from the other delta. For this decision, only the current state reached during the merge and the deltas are available. However, the state in which the future operation will be executed may be relevant, as well. Thus, some conflicts may go unnoticed.

3. MODELS IN EMF

In EMF, each *model* is an instance of an *Ecore model* which defines the types to be instantiated. Ecore models are in turn instances of the *Ecore metamodel*. Figure 3 shows a cutout of the Ecore metamodel as far as it is relevant for the purpose of this paper. Essentially, we have stripped all parts from the full metamodel which do not affect *stored model instances* (packages, operations, and features which are not stored persistently).

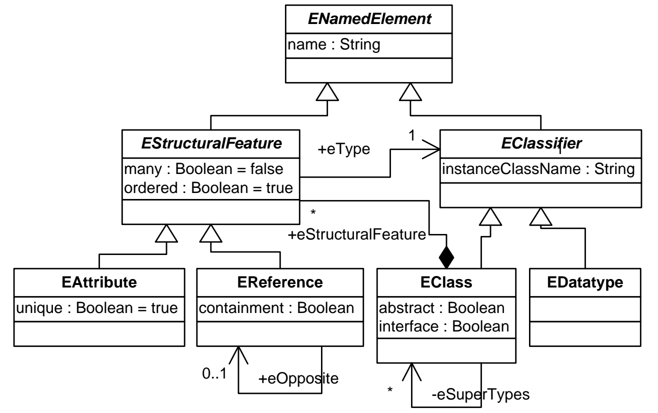

Figure 2: Simplified Ecore metamodel

An Ecore model consists of *classes* which are organized into an inheritance hierarchy. Each class owns a set of *structural features*. With respect to multiplicities, we distinguish only between *single-* and *multi-valued* features (attribute `many`). The Ecore metamodel allows to specify integer values for lower and upper bounds, but the bounds are not checked at runtime. The attribute `ordered` is relevant only for multi-valued features. Features are classified into attributes and references. *Attributes* are typed with *data types*, i.e., Java types such as e.g. `boolean`, `int`, or `String` whose values are considered atomic in EMF. The (meta) attribute `unique` is used to classify multi-valued attributes into *sets* and *bags*. *References* are typed with *EMF classes*. Multi-valued references are always unique. In EMF, references are uni-directional, but two uni-directional references may be declared as *opposite* to represent bi-directional associations. *Containment references* are used to build a tree, which is augmented with *cross references*.

A *model* instantiated from an Ecore model is composed of *objects* instantiated from the respective classes. Each object is composed of (instances of) the features defined for its class in the Ecore model. Each feature may be viewed as a slot for one or more *values*. Please note that in the current EMF implementation multi-valued features are always represented as *lists* even if the order is immaterial. Values of attributes and references are called *data values* and *links*, respectively. A model is *valid* if (1) the containment links form a tree, (2) the target of each link exists (referential integrity), (3) bi-directional links are inverse to each other, and (4) the type and multiplicity constraints of the underlying Ecore model are satisfied.

4. MODEL DIFFERENCES

Our merge algorithm assumes that its input versions have been compared with the help of some *difference algorithm* such as e.g. EMF Compare [7]. In the following, we define the assumptions we make with respect to the outcome of the difference calculation. Any difference algorithm may be used which satisfies these assumptions.

Throughout the rest of this paper, we will always assume that all compared models are of the same type, i.e., they are instances of the same Ecore model.

4.1 State-Based Differences

A *state-based difference* identifies the common and the differing elements of two versions. Common elements are identified by a *matching*. Let v_1 and v_2 denote two model versions and O_1 and O_2 denote their respective sets of objects. An *(object) matching* is a set OM of undirected binary edges between O_1 and O_2. We will write $o_1 \leftrightarrow o_2$ for $\{o_1, o_2\} \in OM$. We require an object matching to be *unique*, i.e., each object may be matched at most once[1]:

$$o_1 \leftrightarrow o_{21} \land o_1 \leftrightarrow o_{22} \Rightarrow o_{21} = o_{22} \tag{3}$$

$$o_{11} \leftrightarrow o_2 \land o_{12} \leftrightarrow o_2 \Rightarrow o_{11} = o_{12} \tag{4}$$

Furthermore, the object matching has to be *type consistent*, i.e., only objects of the same type may be matched. Let $\mathsf{class}(o)$ denote the class from which o was instantiated. Using this function, type consistency is defined as follows:

$$o_1 \leftrightarrow o_2 \Rightarrow \mathsf{class}(o_1) = \mathsf{class}(o_2) \tag{5}$$

Finally, we require *matching roots*. Let r_1 and r_2 denote the roots of v_1 and v_2, respectively. Then the following condition must hold:

$$r_1 \leftrightarrow r_2 \tag{6}$$

For three-way merging, we need three pair-wise matchings on the input versions. Let us consider the union of these matchings. For this union, we require *transitivity*:

$$o_1 \leftrightarrow o_2 \land o_2 \leftrightarrow o_3 \Rightarrow o_1 \leftrightarrow o_3 \tag{7}$$

Concerning *data value matchings*, we assume that only equal data values may be matched. Thus, explicitly stored matchings are not required for single-valued and unordered multi-valued attributes. For ordered multi-valued attributes, we assume that the difference algorithm has calculated a matching which is not necessarily order-preserving.

Virtually all known difference algorithms satisfy the uniqueness Conditions 3–4. Condition 6 is not vital and could be removed easily. Transitivity (7) and type consistency (5) are satisfied if the difference algorithm is based on immutable unique identifiers and the types of objects are immutable, as well. However, they are not necessarily satisfied by algorithms based on similarity values [13]. Violations of Condition 7 may be removed by adding transitive object matchings, provided that the object matchings remain unique. Releasing the requirement for type consistency would require extensions to our merge rules.

4.2 Change-Based Differences

From a matching, a *delta* — a sequence of change operations — could be derived which transforms one of the compared versions into the other one. Although our state-based merge algorithm requires only a matching, it is important to note which kinds of change operations it may handle (Table 1). In addition to assignment, addition, and deletion, our algorithm covers *move operations* because it does not rely on any assumptions regarding the relative positions of matched objects or values.

[1]In the conditions given below, variables are universally quantified.

Category	*Operation*	*Description*
Single-valued feature	$\mathsf{assign}(t, f, v)$	Assign value v to feature f of target t
Multi-valued feature	$\mathsf{remove}(t, f, v)$	Remove value v from feature f of target t
Unordered multi-valued feature	$\mathsf{add}(t, f, v)$	Add value v to feature f of target t
Ordered multi-valued feature	$\mathsf{add}(t, f, v, p)$	Add value v to feature f of target t at position p
Unordered containment reference	$\mathsf{move}(t, c, o)$	Move object o to reference c of target t
Ordered containment reference	$\mathsf{move}(t, c, o, p)$	Move object o to reference c of target t at position p

Table 1: Change operations

5. CONTEXT-FREE MERGING

We will develop the merge algorithm in multiple steps. As a first step, the current section presents a set of *context-free merge rules*. These rules determine the set of objects which should be included into the merged versions and consider each feature of each object without taking the context into account. The interaction among the context-free rules will be discussed in the next section.

5.1 Object Classification

So far, we have assumed that the object sets of the input versions are disjoint. For the following definitions, it is more convenient to deal with overlapping object sets. Based on the matchings between the input versions, the *object identifiers* are re-assigned in such a way that matching objects are assigned the same object identifier. This re-assignment can be performed in a consistent way only if the matchings are unique and transitive. If the matchings are based on unique object identifiers, these identifiers may be used directly, and the re-assignment step may be skipped.

Let O_b, O_1, and O_2 be the object sets of the base versions and both alternative versions. The set of objects O_m of the merged version should include those objects from the base version which have not been deleted from either of the alternative versions, plus the objects which have been inserted on either branch (*object set rule*):

$$O_{ins} = (O_1 \setminus O_b) \cup (O_2 \setminus O_b) \tag{8}$$

$$O_{del} = (O_b \setminus O_1) \cup (O_b \setminus O_2) \tag{9}$$

$$O_m = (O_b \setminus O_{del}) \cup O_{ins} \tag{10}$$

For the following rules, it is necessary to partition the object set of the merged versions into three disjoint subsets. An object in O_{m3} is contained in all input versions and will be processed by a *three-way merge* (11). An object in o_{m2} is contained in both a_1 and a_2, but not in b (duplicate insertion, 12). Even though the overall merge algorithm operates along three ways, only a *two-way merge* may be applied to objects in O_{m2} (i.e., any difference has to be handled as a conflict). Finally, an object in O_{m1} has been inserted into exactly one of the alternative versions (13). Therefore, it has to be *copied* into m.

$$O_{m3} = O_b \cap O_1 \cap O_2 \quad (11)$$
$$O_{m2} = (O_1 \cap O_2) \setminus O_b \quad (12)$$
$$O_{m1} = O_{ins} \setminus (O_1 \cap O_2) \quad (13)$$

5.2 Feature Merging

In this subsection, we give context-free two-way and three-way rules for *feature merging* (and omit the straightforward copy rules for objects from O_{m1}). These rules do not distinguish between attributes and references. In the following, f denotes a feature, the indices b, 1, 2, and m indicate the base version, the alternative versions, and the merged version, respectively. v stands for an arbitrary defined value (either a link or a data value). If different variables are used for values, the values are assumed to be different. The value $\perp$ indicates a conflict which should be resolved by the user.

The following rules determine the values of *single-valued features*[2]. If the values are equal in a_1 and a_2, the value is copied to m (14). If the values are different, but one of them is equal to the base value, the differing value (the change) is selected (15). (16) and (17) handle three-way and two-way conflicts (three and two different values, respectively).

$$f_1(o) = f_2(o) = v \Rightarrow f_m(o) = v \quad (14)$$
$$f_b(o) = v \wedge f_1(o) = v \wedge f_2(o) = v_2 \Rightarrow f_m(o) = v_2 \quad (15)$$
$$f_b(o) = v_b \wedge f_1(o) = v_1 \wedge f_2(o) = v_2 \Rightarrow f_m(o) = \perp \quad (16)$$
$$o \notin O_b \wedge f_1(o) = v_1 \wedge f_2(o) = v_2 \Rightarrow f_m(o) = \perp \quad (17)$$

Although *multi-valued features* are always represented as lists in the current EMF implementation, we make use of the ordered attribute in Ecore models to distinguish between ordered and unordered features. Thus, for unordered features we avoid pseudo merge conflicts on the order of elements.

In the case of *unordered sets*, shared elements are included into m (18). The same applies to elements inserted on one branch (19). Elements deleted on one branch are excluded (20). Three-way merging does not produce any conflicts. Differences between the sets cause conflicts in two-way merging (21)[3].

$$v \in f_1(o) \wedge v \in f_2(o) \Rightarrow v \in f_m(o) \quad (18)$$
$$v \in f_1(o) \wedge v \notin f_2(o) \wedge v \notin f_b(o) \Rightarrow v \in f_m(o) \quad (19)$$
$$v \in f_1(o) \wedge v \notin f_2(o) \wedge v \in f_b(o) \Rightarrow v \notin f_m(o) \quad (20)$$
$$v \in f_1(o) \wedge v \notin f_2(o) \wedge o \notin O_b \Rightarrow (v \in f_m(o)) = \perp \quad (21)$$

Unordered bags are represented as functions which map values into multiplicities. Thus, for some object o a bag-valued feature f is a function $f(o, .) : V \rightarrow \mathbb{N}$, where $\mathbb{N}$ stands for the natural numbers and $f(o, v)$ denotes the multiplicity of v. Insertions and deletions increase and decrease the values of f, respectively. In the case of two-way merging, equal multiplicities are copied to m (22), and a conflict is raised for differing multiplicities (23). For three-way merging, let $df_i(o, v) = f_i(o, v) - f_b(o, v) (i \in \{1, 2\})$ denote the differences in multiplicities in a_i compared to the base version. If both are positive, we take the maximum of the multiplicities in the alternative versions (24). In this way, duplicate insertions are counted only once. Analogously, the symmetric case is handled in which both differences are negative (25). The final rule deals with all other cases (26). By adding both differences to the multiplicity in the base version, insertions and deletions are compensated. No conflict is raised because insertions and deletions commute for bags.

$$o \in O_{m2} \wedge f_1(o, v) = f_2(o, v) = n \Rightarrow f_m(o, v) = n \quad (22)$$
$$o \in O_{m2} \wedge f_1(o, v) \neq f_2(o, v) \Rightarrow f_m(o, v) = \perp \quad (23)$$
$$o \in O_{m3} \wedge df_1 > 0 \wedge df_2 > 0 \Rightarrow f_m(o, v) = \mathsf{max}(f_1(o, v), f_2(o, v)) \quad (24)$$
$$o \in O_{m3} \wedge df_1 < 0 \wedge df_2 < 0 \Rightarrow f_m(o, v) = \mathsf{min}(f_1(o, v), f_2(o, v)) \quad (25)$$
$$o \in O_{m3} \wedge (df_1 \leq 0 \wedge df_2 \geq 0 \vee df_2 \leq 0 \wedge df_1 \geq 0) \Rightarrow f_m(o, v) = f_b(o, v) + df_1(o, v) + df_2(o, v) \quad (26)$$

In the case of *ordered sets*, we apply Rules (18–21) to determine the elements to be included into the merged set. We solve the problem of deriving the order of these elements by *graph algorithms* because the uncoordinated application of feature rules as given above — operating e.g. on a predecessor feature — does not guarantee a linear order. These algorithms aggregate elements into linearly ordered *clusters*. Within each cluster, the order of its elements has to be determined e.g. by user interaction or random ordering.

Since the algorithms for two- and three-way merging are closely related, we provide an integrated description. In Algorithm 1, an operation called *node contraction* is used: All nodes belonging to a *cluster* are replaced with a cluster node, and all adjacent edges from and to nodes outside the cluster are redirected to the cluster node. Furthermore, *node deletion* removes a node and connects all of its predecessors to all of its successors.

ALGORITHM 1 (MERGING OF ORDERED SETS).
Let V_1, V_2, and V_b denote three ordered sets of values (with $V_b = \emptyset$ in the case of two-way merging).

1. Construct graphs $g_1 = (V_1, E_1)$, $g_2 = (V_2, E_2)$, and $g_b = (V_b, E_b)$ each of which consists of a chain of element nodes augmented with a start node and an end node.
2. Contract nodes for shared elements as well as the start and the end nodes.
3. From the resulting graph g, delete all nodes for elements $v_b \in V_b \setminus (V_1 \cap V_2)$.
4. Remove all edges $(v, v') \in E_b \setminus (E_1 \cap E_2)$.
5. Contract nodes on cycles into a cluster node until all cycles are eliminated.
6. Contract sets of nodes which are not mutually related transitively by order relationships.
7. Eliminate transitive edges. (After this step, the graph consists of (cluster) nodes which are arranged in a linear order.)

[2] Symmetric rules are omitted.
[3] The conclusion in (21) states that the assertion $v \in f_m(o)$ has an undefined value.

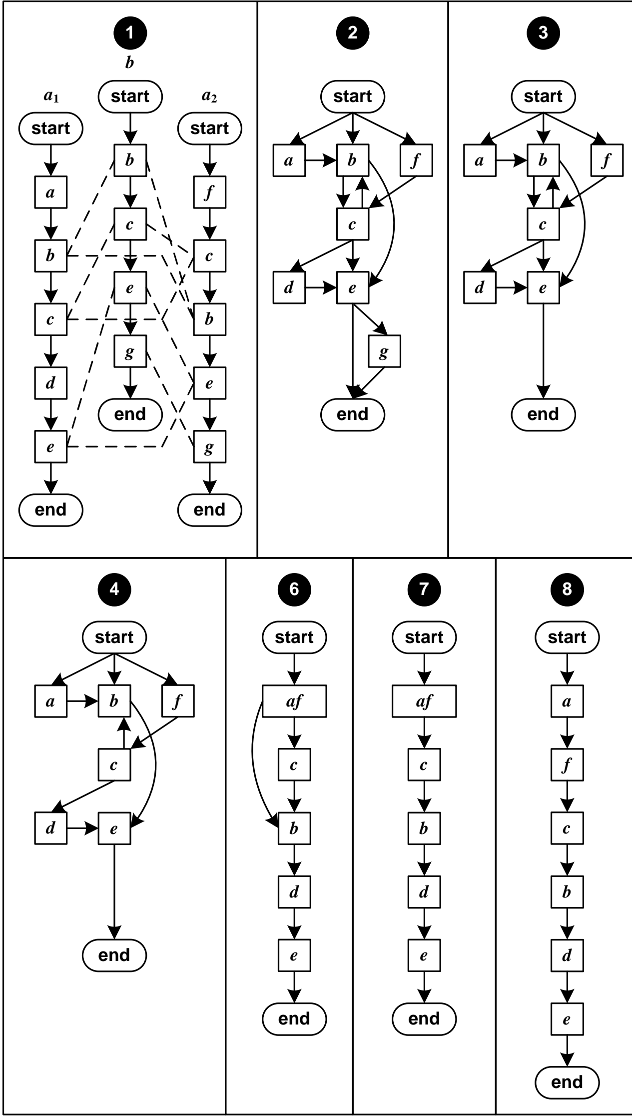

Figure 3: Three-way merging of ordered sets

8. Process the clusters in their linear order. For each cluster C, perform the following steps:
   (a) In the case of two-way merging, determine for each $v \in C \setminus (V_1 \cap V_2)$ whether v should be included into V_m. If not, remove v from C.
   (b) If $C = \emptyset$, delete the cluster node.
   (c) If $C \neq \emptyset$, determine the order of its elements and split up the cluster accordingly.

After Step 8, the graph has a linear order, and each node corresponds to a single element. □

Example 1 (Merging of Ordered Sets). An (abstract) example is given in Figure 3. In Step 1, three linear graphs are created; the base graph is shown in the middle. In Step 2, the graph union is performed. Step 3 deletes g because it is contained in the base and the right alternative, but not in the left alternative. Step 4 removes the edge $c \to e$ because the successors of c in both alternatives are different from the successor in the base; similarly, $b \to c$ is deleted because it is contained in b and a_1 but not in a_2. Step 5 does not have any effect because the graph does not contain cycles. Step 6 contracts a and f because they are not ordered in either direction. Step 7 eliminates the transitive edge $af \to b$. Finally, in Step 8 only the order of a and f still has to be determined. All other decisions have been resolved automatically in Steps 3 and 4. □

Algorithm 1 may be applied also to *ordered bags*. In the case of ordered sets, the matchings which are required as inputs are determined uniquely since each element occurs at most once in each of the sets. In the case of ordered bags, we assume that the matchings have been calculated by some appropriate algorithm (e.g., longest common subsequence, which, however, does not take moves into account).

6. CONTEXT-SENSITIVE MERGING

The context-free rules given in Subsection 5.2 may be applied to attributes as they stand. However, when they are applied to links, they ignore *context-sensitive conflicts*. Therefore, the current section deals with context-sensitive aspects of model merging.

6.1 Structure Graphs

Definition 1 (Structure Graph). For some model *mod*, its *structure graph* $g = (N, E, \mathsf{name}, \mathsf{mult}, \mathsf{inj})$ is a bipartite graph. $N = O \cup R$ is a set of nodes; O and R denote the objects of *mod* and their references, respectively. $R = C \cup X$, i.e., R consists of containment and (non-opposite) cross references, respectively. $E \subseteq (O \times R \cup R \times O)$ is a set of edges (from objects to owned references and vice versa). $\mathsf{name} : N \to \mathsf{string}$ assigns to each object its class name and to each reference the name defined in the Ecore model. $\mathsf{mult} : R \to \{1, *\}$ assigns to each reference its multiplicity. Finally, $\mathsf{inj} : X \to \mathsf{boolean}$ determines whether a reference is injective. □

A *structure graph* constitutes an abstraction of a model which is tailored towards the definition of context-sensitive conflicts and the design of an algorithm for structure merging. The structure graph does not contain attributes and abstracts from the order of links in the case of multi-valued references (which may be determined along the lines of Algorithm 1). Each reference denotes a slot for one or many *links*, which are represented by edges. Please note that opposite references (and their links) are not represented because they may be derived from forward references[4]. The functions name, mult, and inj represent information from the underlying Ecore model which is required for the merge. The names of references are needed to identify the references when building the merged structure graph. The multiplicities of references are required for distinguishing between single- and multi-valued references during the merge. Using the function inj, conflicts on injective references can be detected (36).

Definition 2 (Merged Structure Graph). Let $g_i = (N_i, E_i, \mathsf{name}_i, \mathsf{mult}_i, \mathsf{inj}_i)(i \in \{b, 1, 2\})$ denote structure

[4]The reference class with an incoming EOpposite reference in the Ecore model is arbitrarily defined as the end to be omitted.

graphs for the base version and the alternative versions. The *merged structure graph* is defined as the graph union $g_u = g_b \cup g_1 \cup g_2$, where nodes for objects with the same identifiers as well as nodes for references with the same names are identified. The origins of the elements of g_u are recorded with a function $\mathsf{in} : (V_u \cup E_u) \times I \rightarrow \mathsf{boolean}$, where $I = \{b, 1, 2\}$ is a set of indices, such that:

$$\mathsf{in}(ne, i) = \mathsf{true} \Leftrightarrow ne \in g_i (i \in \{b, 1, 2\}) \quad (27)$$

□

The merged structure graph is an auxiliary intermediate data structure which is used during the merge. This data structure represents the superimposition of the structure graphs of the input versions with the help of interleaved deltas. For each element of the merged structure graph, the function in records in which input versions this element occurs. Eventually, the structure graph of the merged version will be a subgraph of the merged structure graph. To avoid terminological confusion, the former graph will be called the *final structure graph*. The final structure graph has to represent a valid model. In contrast, in general the initial merged structure graph does not correspond to a valid model.

6.2 Context-Sensitive Conflicts

In this subsection, we use the merged structure graph to define *context-sensitive conflicts* with the help of predicate logic.

6.2.1 Containment Conflicts

The *containment trees* of the input models have to be merged in such a way that the merged version has a spanning containment tree. However, the merged structure graph may not be a tree. Containment conflicts defined below indicate violations of the tree structure.

For the definitions referring to containment conflicts, we consider a subgraph of the merged structure graph which contains only containment references $c \in C \subseteq R$ and does not contain graph elements which would be deleted by three-way merge rules. The resulting graph is called *containment graph* and contains only nodes for objects in O_m (10) and C_m (containment references for objects in O_m).

A *containment conflict* is a graph pattern which violates the tree structure of the containment graph. More specifically, a *single-valued containment conflict* occurs on some containment reference $c \in C_m$ if multiple links compete for a single slot:

$$\mathsf{mult}(c) = 1 \wedge (\exists o_1, o_2 \in O_m : o_1 \neq o_2 \wedge c \rightarrow o_1 \wedge c \rightarrow o_2) \quad (28)$$

A *non-unique container conflict* occurs on some object $o \in O_m$ if there are multiple competing containers:

$$\exists c_1, c_2 \in C_m : c_1 \neq c_2 \wedge c_1 \rightarrow o \wedge c_2 \rightarrow o \quad (29)$$

A *cyclic containment conflict* occurs on some object $o \in O_m$ if the object is (transitively) contained in itself:

$$o \xrightarrow{+} o \quad (30)$$

Finally, a *dangling component conflict* occurs on some object $o \in O_m$ if it is not reachable from the root r:

$$\neg(r \xrightarrow{+} o) \quad (31)$$

6.2.2 Delete Conflicts

Context-sensitive conflicts do not necessarily result in inconsistencies. Please recall that we have to detect conflicting changes, regardless whether they result in inconsistencies or not. In particular, deletion of an object invalidates all operations applied to that object. We distinguish between three types of *delete conflicts*: delete-modification, delete-move, and delete-reference conflicts.

An object o is involved in a *delete-modification conflict* if it was deleted from one branch and was modified on the other branch. The term "modification" denotes any change in the subtree with root o. The delete operation invalidates the change operations.

Let us define a delete-modification conflict formally: For some object o, let $\mathsf{tree}(o)$ denote the tree with root o. Furthermore, let $. = .$ be a predicate on trees which yields true if and only if the trees are equal. We define the set of modified objects as follows:

$$O_{mod} = \{o \in O_b | \\ \exists i \in \{1, 2\} : o \in O_i \wedge \mathsf{tree}_b(o) \neq \mathsf{tree}_i(o)\} \quad (32)$$

A *delete-modification conflict* occurs on o if o is classified as both deleted and modified:

$$o \in O_{delmod} = O_{del} \cap O_{mod} \quad (33)$$

A *delete-move conflict* occurs on some object o if o was deleted on one branch and was moved on the other branch. Thus, the object is classified as deleted and has multiple containers in the merged structure graph:

$$o \in O_{del} \wedge \exists c_1, c_2 \in C_u : c_1 \rightarrow o \wedge c_2 \rightarrow o \wedge c_1 \neq c_2 \quad (34)$$

Deleting an object o on one branch and adding a link to it on the other branch raises a *delete-reference conflict*. Please note that the inserted link may refer to o or any component of o in its containment tree ($\xrightarrow{c*}$ denotes the reflective and transitive closure over containment links):

$$o \in O_{del} \wedge \exists x \in X_m, o' \in O_u : o \xrightarrow{c*} o' \wedge x \rightarrow o' \notin E_b \quad (35)$$

6.2.3 Reference Conflicts

In EMF, references are uni-directional, but they may be composed into pairs if bi-directional navigation is required. As already mentioned in Subsection 6.1, opposite references are treated as derived data and therefore are not represented in the merged structure graph. Rather, maintaining consistency of bi-directional links is delegated to the EMF base operations when the merge model is constructed from the final structure graph.

However, there is one property of opposite references which has to be considered during the merge: In the Ecore model, an opposite reference may be declared as single-valued. Then the primary references have to be *injective* (as noted by the function in in the merged structure graph). An *injectivity conflict* occurs on two cross references x_1, x_2 if the references are marked as injective and their inverse links refer to the same object:

$$x_1 \in X_m \wedge x_2 \in X_m \wedge x_1 \neq x_2 \wedge$$
$$\mathsf{name}_u(x_1) = \mathsf{name}_u(x_2) \wedge \mathsf{inj}_u(x_1) \wedge \mathsf{inj}_u(x_2) \wedge$$
$$\exists o \in O_m : x_1 \rightarrow o \wedge x_2 \rightarrow o \quad (36)$$

EXAMPLE 2 (CONTEXT-SENSITIVE CONFLICTS). Part a of Figure 4 shows a merged structure graph which will be used later for demonstrating the context-sensitive merge algorithm. Colors and line styles indicate in which input versions the elements are contained. A *single-valued containment conflict* occurs on o_1 because the original component o_2 was replaced with the new component o_{10} on one branch and the already existing component o_5 was moved to the container o_1 on the other branch. A *non-unique container conflict* occurs on o_5 because o_5 was moved to different containers (o_1 and o_8, respectively). A *cyclic containment conflict* occurs on o_3 and o_4 because o_3 was moved to o_4 and vice versa. A *dangling component conflict* occurs on both o_3 and o_4 because they lost the connection to the root r. In addition, the same type of conflict occurs on the new component o_9 because its container o_6 was deleted on the other branch. A *delete-modification conflict* occurs on o_6 because it was deleted one branch, while the new component o_9 was added on the other branch. A *delete-reference conflict* occurs on o_7 because it was deleted from one branch and a cross link from o_4 was added on the other branch. This conflict is propagated upwards to o_6 because deletion of o_6 would imply the deletion of o_7. Finally, an *injectivity conflict* occurs on the cross links emanating from o_8 and o_9 which were added on different branches and refer to the same object o_3. □

6.3 Algorithm for Context-Sensitive Merging

The following algorithm starts with constructing the merged structure graph, which is successively transformed into the final structure graph. First, the containment tree is built; subsequently, cross links are processed. Conflicts are detected and resolved on the fly. The final structure graph is a representation of the structure of a valid model.

While we have specified rules for the detection of context-sensitive conflicts in a declarative way in the previous subsection, we follow an operational approach to the actual construction of the final structure graph. The control structure of the graph algorithm is required to coordinate the resolution of inter-dependent context-sensitive conflicts in such a way that the result of the merge is a valid model.

ALGORITHM 2 (CONTEXT-SENSITIVE MERGING). Let g_b, g_1, and g_2 denote the structure graphs of the input versions. Context-sensitive merging is performed by constructing the merged structure graph and iteratively removing elements which are not be included into the final structure graph:

1. *Initialization*
   (a) Construct the merged structure graph
   $$g_u := g_b \cup g_1 \cup g_2 \quad (37)$$
   (b) Insert the containment references of the root into the set C:
   $$C := \{c \in C_u | r \rightarrow c\} \quad (38)$$
2. *Build the containment tree* top-down, taking context-sensitive conflicts into account. While $C \neq \emptyset$, do:
   (a) Select and remove some $c \in C$.
   (b) Determine the targets of all links emanating from c:
   $$T := \{o \in O_u | c \rightarrow o \in E_u\} \quad (39)$$
   (c) Apply context-free merge rules for containment links to all objects $o \in T$ which are not involved in a context-sensitive conflict.
   (d) For objects $o \in T$ involved in conflicts, resolve the conflict to decide whether o is to be included into the component set.
   (e) Let $T' \subseteq T$ denote the set of components selected from the target set. Remove obsolete containment links not ending in T' as well as conflicting containment links which emanate from other containment references and end at objects in T':
   $$E_u := (E_u \setminus \{c \rightarrow o | o \notin T'\}) \setminus \{c' \rightarrow o | c \neq c' \wedge o \in T'\} \quad (40)$$
   (f) Purge all objects which are no longer reachable from the root r via containment links from g_u.
   (g) For each $o \in T'$ which was involved in a delete conflict, propagate the undelete operation to composed nodes and edges. Let $\mathsf{in}(o, i) = \mathsf{false}$ for some $i \in \{1, 2\}$. Perform a transitive closure over containment links, regarding only nodes or edges ne which were contained in b but not in a_i. Undelete is performed by updating the function in for o and all transitively reached ne:
   $$\mathsf{in}(o, i) := \mathsf{true} \quad (41)$$
   $$\mathsf{in}(ne, i) := \mathsf{true} \quad (42)$$
   (h) Add the containment references of all objects in T' to C:
   $$C := C \cup \{c' \in C_u | \exists o \in T' : o \rightarrow c'\} \quad (43)$$
3. *Process cross links* $x \in X_u$ as follows:
   (a) Determine the target set of all cross links emanating from x:
   $$T := \{o \in O_u | x \rightarrow o \in E_u\} \quad (44)$$
   (b) Apply context-free merge rules for links to all objects $o \in T$ which are not involved in a context-sensitive conflict (an injectivity conflict).
   (c) In the case of an injectivity conflict of $x \rightarrow o$ with some $x' \rightarrow o$, resolve the conflict by selecting one of the links and deleting the competing link.
   (d) Let $T' \subseteq T$ denote the set of objects selected from the target set T. Delete links to objects outside of T':
   $$E_u := E_u \setminus \{x \rightarrow o | x \notin T'\} \quad (45)$$

□

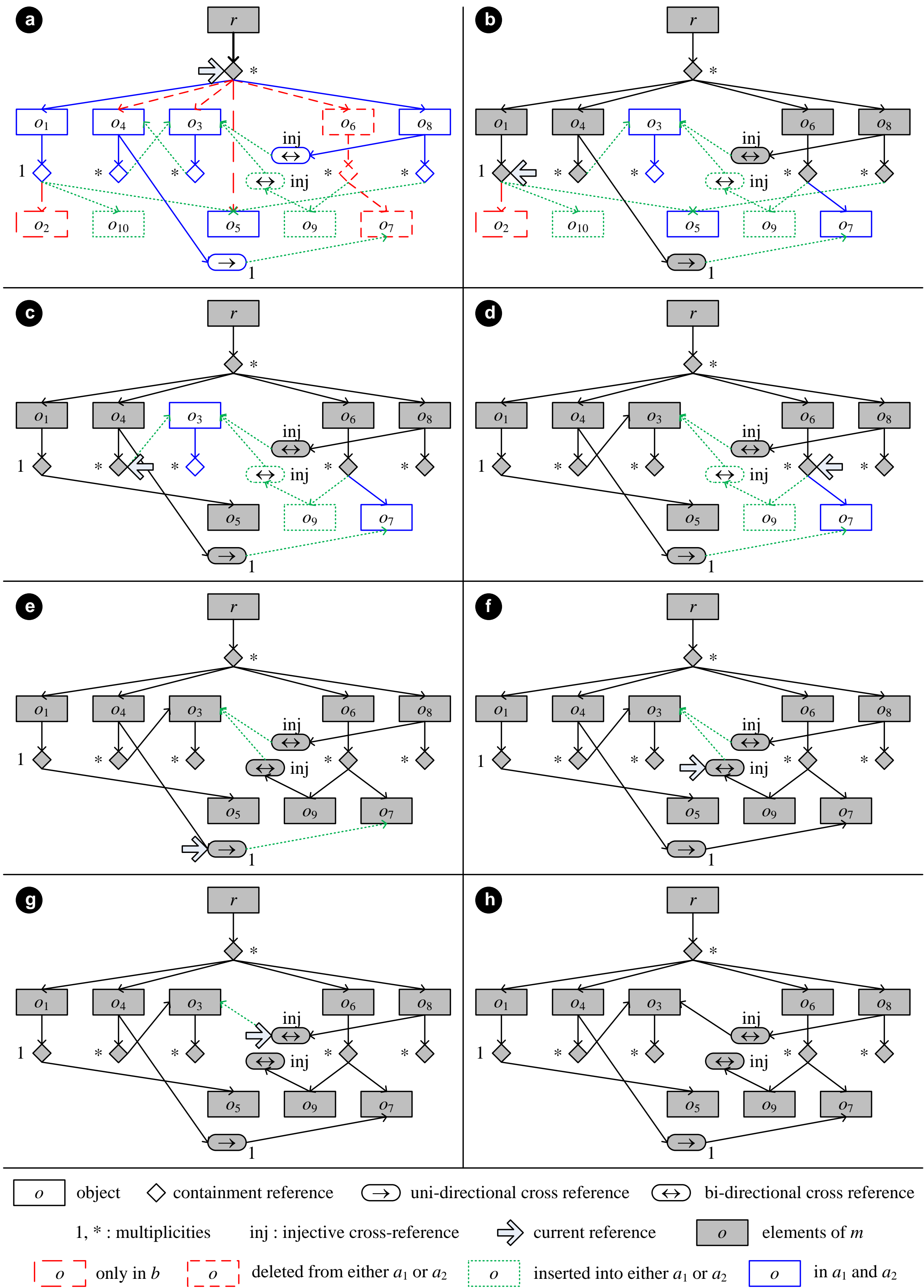

Figure 4: Example: Context-sensitive merging

EXAMPLE 3 (CONTEXT-SENSITIVE MERGING). Figure 4 illustrates how the merged structure graph which we have introduced in Example 2 is transformed into the final structure graph. Each subfigure displays a state of the graph reached during the execution of the algorithm. In each state, the elements which have already been inserted into the final structure graph are displayed in gray. Furthermore, the reference to be processed is indicated by an arrow. State a is reached after the initialization and the selection of the (only) containment reference of the root. States a–d belong to Step 2 of the algorithm, in which the containment tree is built. States e–g are reached during the execution of Step 3 (processing of cross links). State h is the final state.

In *State a*, the containment reference of the root — say c — is processed. The link to o_5 is deleted because it is contained in neither of the alternative versions. o_1 and o_8 are inserted into m since they are shared components. Although the links to o_3 and o_4 were removed from the alternative versions, they are offered as candidates because both are involved in context-sensitive conflicts (a cyclic containment conflict and dangling component conflicts). We assume that it is decided to include o_4 but not o_3. Since each object must have at most one container, the containment link from o_3 to o_4 is removed, which eliminates the cyclic containment conflict. Finally, a delete-modification conflict occurs on o_6. We assume that that this conflict is resolved by giving preference to the modifications. Therefore, the containment link to o_6 is inserted into m, and an undelete operation is applied to o_6 and its contained components. As a consequence, o_7 is re-classified as a shared element.

In *State b*, the containment reference of o_1 is processed. The link to o_2 is deleted because it belongs to the base only. Now, o_2 is unreachable and is purged from the graph. A single-valued containment conflict occurs on the reference. We assume that the link to o_5 is selected. Therefore, o_5 is inserted into m, and the conflicting containment link from o_8 is deleted. Furthermore, the link to o_{10} is removed, which now becomes a dangling component and is removed even though it was classified as an inserted element.

In *State c*, processing continues with the containment reference of o_4: The link to o_3 as well as its target are added to m.

In *State d*, the containment reference of o_6 is processed. We observe that o_7 has previously been re-classified as a shared element. Therefore, it is inserted into m. Without the re-classification, o_7 would have (erroneously) been deleted. o_9 is inserted as a new element. The next step (processing of o_8) is skipped because it does not change g_u. This terminates the phase for building up the containment tree and leaves us with the cross references, which still have to be processed.

In *State e*, the link from o_4 to o_7 is inserted. Please note that in the initial state of g_u a delete-reference conflict occurred on o_7. In the meantime, however, the deletion of o_7 has been undone, eliminating the delete-reference conflict. Therefore, the link can now be inserted automatically.

In *State f*, the cross reference from o_9 is processed. An injectivity conflict with the reference emanating from o_8 is detected. We assume that the reference from o_8 is selected. As a consequence, the conflicting reference from o_9 is deleted.

In *State g*, the link from o_8 to o_3 is inserted automatically because the injectivity conflict has already been resolved in the previous step.

In *State h*, we obtain the final structure graph. The graph represents the structure of a valid model and incorporates all non-conflicting changes, as well as those alternatives which have been selected by the user in the case of context-free or context-sensitive conflicts. □

7. PROPERTIES

The overall merge algorithm may be synthesized from the parts given in Sections 5 and 6 by extending the structure graph and the merged structure graph with attributes and orderings, using Algorithm 2 to determine the containment tree and the cross links of the merged version, applying the context-free rules for feature merging given of Subsection 5.2 to attributes, and using Algorithm 1 for determining the order of set-valued features (both attributes and links). In the following, we state two properties of the overall merge algorithm.

THEOREM 1 (VALIDITY OF THE MERGED VERSION). Let b, a_1, and a_2 be valid models of the same type t which are connected by consistent pair-wise matchings satisfying Conditions 3–7. Under these prerequisites, the constructed merged version m is a valid model of type t.

PROOF. Since the input versions are valid models of the same type and identified objects are instances the same class, each object in m has a unique class and the features defined for this class. For all features of all objects in m, the values are instances of types defined in the Ecore model because the values in m have already occurred in the input versions. The context-free rules for feature merging guarantee that the values of single-, set- and bag-valued features are single values, sets, and bags, respectively. For ordered set-valued features, Algorithm 1 constructs a *linear order*.

m has a spanning tree: Step 2 of Algorithm 2 constructs the containment tree top-down. Each (non-root) object is added to the containment tree together with one incoming containment link; competing incoming containment links are deleted. Furthermore, objects which are not reachable are deleted. Referential integrity is maintained because in Step 3 of Algorithm 2 cross links are created only to objects in m. Finally, consistency of bi-directional links is guaranteed by considering only one direction explicitly during the merge and delegating the creation of opposite links to the EMF base layer. □

THEOREM 2 (DETECTION OF CONFLICTS). Let b, a_1, and a_2 be consistent models of the same type t which are connected by consistent pair-wise matchings satisfying Conditions 3–7. Let Δ_1 and Δ_2 be sequences of operations calculated from these matchings such that $a_i = \Delta_i(b), i \in \{1, 2\}$. Under these prerequisites, the merge algorithm detects all change-based conflicts between operations in these deltas.

PROOF. According to Conditions 1–2, two operations $op_i \in \Delta_i, i \in \{1, 2\}$ are in conflict if one of them invalidates the other one or they do not commute. The types of operations to be considered were listed in Table 1. For each pair of types, we have to show that our state-based merge algorithm detects change-based conflicts with the help of context-free and context-sensitive merge rules. This proof goes beyond the scope of this paper; see [20]. □

8. RELATED WORK

Operation-based merging [16] tries to solve the merge problem at a very generic level by merging operation sequences. The algorithm may be applied to any abstract data type, provided that decision procedures for the commutativity of operations have been written. A conflict occurs if operations do not commute. To detect such conflicts, all interleavings of the operation sequences have to be considered. The merge algorithm is not only computationally expensive. In addition, the merge is complex to handle for the user because different interleavings may results in different states which all have to be considered.

In [12], an operation-based tool for three-way merging is presented which may be applied to models which are represented as graphs. In contrast to operation-based merging described above, a fixed metametamodel is assumed (Rationale Based Unified Software Engineering Model). Conflict analysis is performed using only the operation sequences which transform a common base version into two alternative versions. This reduces the complexity of analysis, but constrains the set of detectable conflicts: Only the operations and their actual parameters, but not the states in which they are executed are taken into account. Merging is an interactive process where the user has to decide for each operation whether it shall be applied or skipped. If an operation is selected, all required operations from the same sequence are selected, as well, and all conflicting operations from the other sequence are deselected.

[18] presents an approach to three-way merging of models represented as graphs which is based on category theory. With the help of pushouts, a union graph is constructed which contains all elements from all input version. The union graph resembles the merged structure graph in our approach. In the union graph, elements are marked as created, common, deleted, renamed, or moved. Furthermore, conflicts are detected with the help of generic rules which may be augmented with specific rules taking the constraints of a specific metamodel into account. For example, there is a generic rule to detect delete-reference conflicts. Unfortunately, conflicts are only detected, but not resolved: The merge process stops when a conflict is flagged in the union graph. Furthermore, the approach operates at a more general level than our work and would have to be adapted for merging of models in EMF.

[6] presents a framework for model merging which is based on algebraic specifications. A generic merge operator may be applied to models of any type defined by an algebraic specification. The operator performs a union of two models, identifying common elements. That is, it performs two-way merging, where all differing elements are included automatically into the merged version. This approach resembles schema integration in database management systems, but differs considerably from three-way merging.

[5] proposes a generic three-way merge algorithm which is applied to models at a low level of abstraction. The result produced by the merge may be inconsistent, both with respect to the respective meta model and the underlying metametamodel. In a post-processing phase, inconsistencies are detected automatically and resolved interactively. In our approach, the model merge produces a valid model, i.e., fundamental inconsistencies which may prevent further processing of the merge result are avoided. However, violation of specific constraints — e.g., expressed in OCL in addition to an Ecore model — still requires post-processing.

In [19], we have presented a tool for three-way merging of software documents. The tool merges software documents of arbitrary types which are represented as abstract syntax graphs. The merge is based on a pair-wise comparison of the input versions (state-based merging). It preserves context-free correctness, detects context-free two- and three-way merge conflicts, and detects and partially resolves context-sensitive conflicts by a post-processing phase. The work presented in this paper differs from our previous work in two respects: First, the underlying data models are different. Second, the old algorithm takes neither deletion conflicts nor move operations into account. Both extensions require a considerably more sophisticated merge algorithm.

3dm [15] is a tool for three-way merging of XML documents. The tool focuses on the tree structure and does not consider links (non-hierarchical relationships). The merge tool assumes mappings among the inputs versions from which the changes are derived (including move operations). For the merge, all input versions are represented as sets of facts (tuples for representing both the structure and the attributes of the respective XML documents). These sets are united, and obsolete facts from the base are removed. The resulting set is analyzed for conflicts. A conflict occurs if a single-valued feature is assigned multiple values. This way of reasoning essentially corresponds to our context-free merge rules. Apart from delete-modification conflicts which can be recognized in a post-processing phase, context-sensitive conflicts are not taken into account. Furthermore, it cannot be guaranteed that the resulting set of facts corresponds to a well-formed XML document (even if no conflicts are detected).

[17] presents a framework for merging models based on given correspondences. The framework is based on a metametamodel supporting different kinds of semantic relationships. The merge rules guarantee consistency with respect to this metametamodel, but they would not guarantee valid models in EMF. The framework may be adapted, but implementing the semantics required for EMF models is not straightforward.

In [1], a change-based algorithm for three-way merging of MOF model instances is presented. The algorithm handles a subset of the MOF metametamodel which is similar to Ecore. It is assumed that model elements are assigned unique identifiers. Furthermore, it is assumed that no model element is inserted twice, i.e., if Δ_1 creates an object with unique identifier u, Δ_2 cannot create another object with the same identifier u. Even with the use of unique identifiers, this assumption may not be satisfied if a_1 and a_2 are not merged with respect to their most recent common ancestor. Furthermore, the algorithm cannot handle move operations. The produced output may be inconsistent with respect to the MOF metametamodel. Containment conflicts may go unnoticed, resulting e.g., in containment cycles or non-unique containments. Furthermore, merging two ordered sets may result in an ordered bag.

[9] describes a tool for three-way merging of MOF model instances which was developed in the MOFLON project [4]. The tool was inspired by the work presented in [1], but follows a state- rather than a change-based approach. Apart from this difference, the comments given above apply also to the tool implemented in MOFLON.

EMF Compare [7] is a differencing tool which compares two versions of a model which may be instantiated from an

arbitrary Ecore model. The tool can also be used for an interactive two-way merge, where the user removes the differences between the versions step by step. A three-way merge may also be performed with EMF Compare. In this case, the tool tries to detect conflicts among competing changes. Some conflicts (e.g., delete-modification conflicts or conflicting moves of the same element) are found, while others go unnoticed (e.g., delete-reference conflicts or cyclic moves). Furthermore, EMF Compare cannot merge ordered sets.

EMF Compare is used in the Rational Software Architect (RSA) for comparing and merging UML models [14]. Internally, both semantic and notational models are represented as EMF models. Physical changes on the EMF models are mapped back to logical changes on the respective UML models. Thus, the RSA merge tool operates at a different (higher) level of abstraction as EMF Compare itself and the approach presented in this paper. On the (more specific) level of UML models, RSA provides several mechanisms for ensuring a consistent merge result (called model integrity protection). Our work targets at improving merge support at the EMF level, thereby complementing the UML-specific extensions of RSA.

While the RSA merge tool still operates essentially on a syntactic level (of UML models, particularly UML class diagrams), SMoVer [2] goes a step further and addresses merging at a semantic level. It is argued that syntactic merging may produce conflicts even though the models to be merged are equivalent at the semantic level. This requires reasoning capabilities which are specific to the respective metamodel. This level of reasoning is not addressed by our approach.

To specify our algorithm for three-way merging, we have resorted to rather classical aids such as logical formulas and graph algorithms. For comparing and merging models, several frameworks have been developed for the specification of merge rules, conflict detection, and conflict resolution [10, 8]. The work presented in this paper might serve as an interesting test case for these frameworks.

9. CONCLUSION

We have presented a formal approach to three-way merging of models in the EMF framework which produces a valid model, handles move operations, and detects and resolves context-free and context-sensitive conflicts. Previous approaches satisfy these properties at best partially. The formal specification serves as a foundation for developing a three-way merge tool which implements the rules and algorithms given in this paper.

10. REFERENCES

[1] M. Alanen and I. Porres. Difference and union of models. In *UML 2003*, LNCS 2863, pages 2–17, San Francisco, CA, 2003. Springer-Verlag.

[2] K. Altmanninger, W. Schwinger, and G. Kotsis. Semantics for accurate conflict detection in SMoVer: Specification, detection and presentation by example. *International Journal of Enterprise Information Systems*, 6(1):68–84, 2010.

[3] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[4] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A standard-compliant metamodeling framework with graph transformations. In *ECMDA-FA 2006*, LNCS 4066, pages 361–375, Bilbao, Spain, 2006. Springer Verlag.

[5] C. Bartelt. Consistence preserving model merge in collaborative development processes. In *CVSM 2008*, pages 13–18, Leipzig, Germany, 2008. ACM Press.

[6] A. Boronat, J. A. Carsí, I. Ramos, and P. Letelier. Formal model merging applied to class diagram integration. *Electronic Notes of Theoretical Computer Science*, 166:5–26, 2007.

[7] C. Brun and A. Pierantonio. Model differences in the Eclipse modelling framework. *UPGRADE*, IX(2):29–34, Apr. 2008.

[8] A. Cicchetti, D. D. Ruscio, and A. Pierantonio. Managing model conflicts in distributed development. In *MoDELS 2008*, LNCS 5301, pages 311–325, Toulouse, France, 2008. Springer-Verlag.

[9] R. Deußer. 3-Wege-Merge auf Instanzen MOF-konformer Metamodelle. Master's thesis, Technical University of Darmstadt, Darmstadt, Germany, 2008.

[10] K.-D. Engel, R. F. Paige, and D. S. Kolovos. Using a model merging language for reconciling model versions. In *ECMDA-FA 2006*, LNCS 4066, pages 143–157, Bilbao, Spain, 2006. Springer-Verlag.

[11] S. Förtsch and B. Westfechtel. Differencing and merging of software diagrams - state of the art and challenges. In *ICSOFT 2007*, pages 90–99, Barcelona, Spain, 2007.

[12] M. Kögel. Towards software configuration management for unified models. In *CVSM 2008*, pages 19–24, Leipzig, Germany, 2008. ACM Press.

[13] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, and R. F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *CVSM 2009*, pages 1–6, Vancouver, BC, Canada, 2009. IEEE Computer Society Press.

[14] K. Letkeman. *Comparing and Merging UML Models in IBM Rational Software Architect: Part 3*. IBM, Aug. 2005.

[15] T. Lindholm. A three-way merge for XML documents. In *ACM Symposium on Document Engineering*, pages 1–10, Milwaukee, WI, 2004. ACM Press.

[16] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE5*, pages 78–87, Tyson's Corner, Virginia, Dec. 1992. ACM Press.

[17] R. A. Pottinger and P. A. Bernstein. Merging models based on given correspondences. In *VLDB 2003*, pages 862–873. VLDB Endowment, 2003.

[18] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A category-theoretical approach to the formalisation of version control in MDE. In *FASE 2009*, pages 64–78, York, UK, 2009. Springer-Verlag.

[19] B. Westfechtel. Structure-oriented merging of revisions of software documents. In *SCM-3*, pages 68–79, Trondheim, Norway, 1991. ACM Press.

[20] B. Westfechtel. Three-way merging of EMF models. In preparation, 2010.