History-based Merging of Models

Maik Schmidt, Sven Wenzel, Timo Kehrer, Udo Kelter
Software Engineering Group
University of Siegen
{mschmidt,wenzel,kehrer,kelter}@informatik.uni-siegen.de

Abstract

Model-driven development in collaborative teams requires services for merging models. Such services should have the same quality as one is used to for source code. The constraints which are relevant in model driven engineering imply that the merging of models must be implemented differently than merging of texts. Based on the analysis of these constraints, we present an approach for merging models that attains a high level of consistency of the merged models and minimizes the loss of effort due to conflict resolution.

1 Introduction

Model-driven engineering increasingly becomes common practice in software development. Complex models are most often collaboratively developed in teams. One of the key problems of collaborative work on shared documents is the merging of parallel changes or different development branches. Thus, it is essential to have merge tools for models that offer the same functions in much the same quality as it is available for textual documents.

Techniques for model merging must consider the conditions under which models are edited and used in development processes, notably the technical characteristics of tools which process models. These conditions differ significantly from those which are valid for texts. Section 3 analyzes the resulting requirements for merge tools and shows that processes commonly used for merging textual documents cannot be simply applied to models. We also point out that the level of abstraction on which models are represented is crucial for merging techniques.

Section 4 discusses existing solutions and their limitations. With this background, Section 5 presents a new approach for model merging which guarantees a high level of consistency of merge results and at the same time minimizes the loss of development effort due to conflict resolution.

2 Basic terminology

Generally, merging denotes the process of combining a set of documents into one consolidated document. We generally assume that the documents involved in a specific merge process are of the same type. A **three-way merge** function takes two documents, D1 and D2, which are variants of each other and must both be revisions of a common base version D0. The merge result, i.e. the output of a three-way-merge, is a new document which contains all changes from D0 to D1 and from D0 to D2, respectively. The changes from D0 to D1 and D2 can be determined in an arbitrary manner, for example by logging editing commands in model editors. More commonly used approaches are the calculation of an asymmetric difference, i.e., a sequence of operations [3], from D0 to D1 and D2, respectively. An important objective is to preserve all work involved in the changes to the base version.

Edit data type We assume that documents are edited by means of a set of document-type specific operations and that these edit operations and their semantics are defined by an abstract data type. We refer to this abstract data type as the **edit data type** of the considered class of documents. The set of operations in a given edit data type must be rich enough to construct any possible document state. Typical operations are the insertion, deletion, modification, etc. of document elements. The edit data type can contain further operations, for example operations which move document elements to a different location. Such operations can be called complex since their effect can also be achieved using several elementary operations. Complex operations allow us to express the difference between two documents more compactly. However, some external representations of differences cannot handle complex operations. Section 3.1 discusses the choice of the edit data type in greater detail.

Merge conflicts and merge decisions A pair of changes, from D0 to D1 and D0 to D2, may be technically or conceptually incompatible. In case of a technical incompatibility, at least one of the possible sequences of edit operations would create an error and result in a failure. In case of a conceptual incompatibility, applying both changes would result in a document state violating some document-type specific consistency or integrity constraints. Such a pair of changes is denoted as a **(merge) conflict**. Changes that are not in conflict can be mutually adopted and integrated into the merge result. In case of a conflict, however, one of the conflicting changes has to be excluded from the merge result. A **merge decision** is required here. There are two different approaches to taking merge decisions:

1. **Non-interactive merging:**
   Non-interactive approaches initially create a consolidated intermediate document which comprises all the changes within *both documents* D1 and D2. In case of conflicts, the relevant document elements have to be either duplicated, or some additional elements that neither belong to D1 nor to D2 have to be created as *auxiliary data*. The automatic merge function of RCS (or CVS and SVN), for instance, is a representative of a non-interactive merging approach. Move operations or other complex operations cannot usually be represented by these merging approaches, although complex changes would generally be more comprehensive.

   Due to the inclusion of auxiliary data or conflicting changes, the initially created intermediate document usually is inconsistent and must be manually corrected on the basis of relevant merge decisions. Hence, non-interactive approaches require further editing of the intermediate document. However, none of the conflicting changes has to be re-edited from scratch, and both change variants can be modified in order to be integrated into the final merge result.

2. **Interactive merging:**
   Interactive merging approaches ask the developer for merge decisions *during* the merge process. Possible merge decisions are to adopt the changes according to D1 or according to D2, none of the changes, or modified changes. This approach supports the developer in creating a merge result which is a consistent document.

   The main disadvantage of such an interactive approach is that merge tools have to be realized individually for each document type. Three-way-merge tools are considerably more complex than pure difference tools [3] or 2-way-merge tools, which can be relatively easy derived from difference tools.

3 Particularities of Models

In practice, non-interactive merging approaches are dominating in the case of textual documents. However, methods which are well established for textual documents cannot be directly applied to models. In the following, we outline the principal problems.

3.1 Model Representations

Merge tools for textual documents are often characterized as *generic* because they process documents of various types (such as source code documents of different programming languages, LaTeX source documents etc.). This view of merge tools for textual documents is incorrect insofar as those merge tools actually are based on a single document type, which is merely text. Text consists of a sequence of lines, which in turn consist of a sequence of characters. The *edit data type of texts* simply comprises operations to insert or delete lines into or from a text. Optionally, an edit data type for texts might include operations to insert or delete sequences of characters into or from lines. Moreover, such an edit data type is the conceptual basis of simple textual editors which do not consider the syntax of different document types. Such editors are appropriate to edit and correct syntactically incorrect documents so that they can be further processed, by a compiler, for instance.

Textual editors ignoring syntactical constraints play a central role in conventional merge processes. Documents representing source code or similar complex contents are not merged according to their actual document type but on the basis of their low-level textual representations. The merge result of two syntactically correct documents is, most often, a syntactically incorrect document that has to be manually corrected. In other words, the initial merge result, which can also be regarded as a preliminary merge result represented by an intermediate document, resides on a lower consistency level than the input documents. Working on different levels of consistency is very common for documents with a human readable textual representation, but cannot be applied to models. The following representations and their respective edit data types can be distinguished:

1. Textual representations, for example in XML-syntax or other proprietary formats.

2. Representation as a context-free syntax graph of which nodes and edges are specified by a meta-model.

3. Representation as a syntax graph which additionally respects a set of context-sensitive constraints.

4. Representation as a context-sensitive and semantically correct syntax graph; such edit data types would guarantee models that are consistent in form and content. For example, behaviour models that are always compilable are needed in a model-driven development.

Merge functions based on pure textual representations potentially create merge results which violate some basic syntactical restrictions, and thus must be corrected by means of textual editors. This is very error-prone and tedious work, and not applicable to practical model-driven engineering.

Merge functions should produce results which can be processed by model editors. Such editors depend on syntactically correct representations of the models. At minimum, a context-free syntactical correctness, and to some extent, even a context-sensitive syntactical correctness is required in order to display the models graphically on the graphical user interface. Many modelling tools guarantee an even higher level of consistency (as for example they prevent dangling references, enforce uniqueness of identifiers etc.).

Non-interactive merge functions cause big problems in this context. Conflicting change operations cannot be represented in merged documents without additional syntactical concepts indicating the conflict, the reasons for that conflict and possible solutions. Thus, the meta-model must be extended. Due to the changes to the meta-model, the merge result cannot be processed by standard model editors. Hence, low-level editors must be used, which is hardly acceptable.

Merge functions preserving semantics are obviously only applicable to those types of models for which a formal semantics actually exists, and must be realized specifically for each model type. However, many model types do not have formally defined semantics. Further, incomplete documents often occur during development processes. They cannot be sensibly interpreted, but nonetheless, they need to be merged. Therefore, in this paper we will not further consider semantic based merge functions.

3.2 Conflicts

In principle, the set of possible pairs of conflict candidates is extremely huge, as each change in document D1 can be in conflict with any change in D2. In the case of textual documents, this set is radically reduced by the following assumption: changes at significantly different document positions are generally classified to be conflict-free. This assumption is correct in 99% of all situations. The remaining situations, i.e., pairs of changes that are misleadingly recognized as conflict-free, are mostly leading to compiler errors and can therefore be quickly detected.

This approach is not suitable for models, as the concepts of locality and distance known from textual documents cannot be transferred to models. The layout, which could be used as a distance metric, is conceptually irrelevant in terms of model merging. A distance metric based on the abstract syntax graph is not very well-suited either. Here, completely new concepts have to be developed.

Models differ from texts in another aspect: a change in model D1 can be in conflict with many distributed changes in model D2. Similar situations cannot occur with texts: here, only some lines in D2 that are close to the given change in D1 can cause a conflict. These lines can be simply arranged to an entire text block. Such a clustering of changes is not possible in the case of models.

4 Existing Solutions

Merging models can be regarded as similar to the merging of source code documents which are considered as abstract syntax graphs. However, these algorithms [7] are strongly geared to the structures of source code and are barely adaptable to models with respect to the characteristics of syntax trees, the notion of merge conflicts, etc. Other approaches address the problem of model merging by introducing specific model repositories [8]. However, the 'merge' functions presented there are actually offering typical patch functionality.

A common practice in contemporary model-driven engineering is to strictly avoid parallel development and thus situations that require the merging of models [1] as only a few practically usable tools are ready to use for merging models. Two of these tools will be introduced in the remainder of this chapter.

4.1 Matlab/Simulink

The Graphical Merge Tool of Matlab/Simulink is a graphical difference tool for Matlab/Simulink models which incorporates rudimentary merge functions. Two models that are to be compared are shown in two different representations; once as graphical diagrams and once as object trees (cf. Figure 1). The differences are shown in the object tree, where they can be selected. Differences can either be transferred from one model into the other or be discarded. The merge process is a typical 2-way-merging. The human interactive selection of each single difference is a highly tedious and often error-prone work.

4.2 Rational Software Architect

The difference component integrated into Rational Software Architect [2] is one of the most advanced merging tools which are currently available. Besides the pure presentation of differences, both 2-way- and 3-way-merging is supported.

In the graphical representation , the base document, the variants to be merged and the preliminary merge result are each displayed in their usual notation. Moreover, a list of all merge conflicts is presented. Coherent changes, as for example changes affecting the same model element, are arranged in groups.

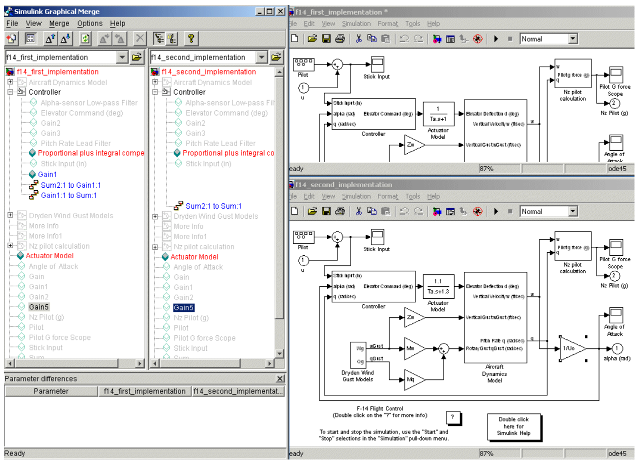


Figure 1. The Graphical Merge Tool of Matlab/Simulink

Merge decisions have to be taken interactively. Each conflict requires choosing one of the possible changes. After each merge decision, a preliminary merge result is created and presented in the usual notation. With respect to groups of conflicts, it is possible to choose the changes of one or the other document, which usually reduces the total amount of merge decisions significantly. Rational Software Architect cannot include two changes that are contradictory in the sense that they would result in a technical or conceptually inconsistent model. Moreover, auxiliary data cannot be integrated into the merge result.

5 History-based Merging of Models

This section presents an approach for 3-way-merging which uses the edit operations that lead to the given state of a model rather than referring to the current states of the models.

5.1 Transactions

Editing activities within a development branch are classifiable in the sense that they can be arranged in groups that correspond to specific development tasks, such as correcting a defect or introducing a new function. Such changes transform the model from one consistent state into another. We call such a group of edit operations as a transaction. The basic idea of transaction-oriented merging of models is to treat transactions, i.e. sequences of changes, as data objects of their own right; they can be edited and saved persistently (if product liability conditions are strict, as e.g. in the automotive domain, each development activity must be accurately documented anyway).

When models are stored within a version management system, the changes between two revisions can roughly be considered as a single transaction. When the version management system is used in a disciplined manner, the commit message documents the semantic purpose of the transaction. The merge tool should offer functions to combine or split up transactions, for example, for a case when several functional changes have been made within a revision.

Principally, the edit operations contained in a transaction can be determined through a comparison of the two revisions [4]. In order to correctly identify model elements across the version history, the entire version history must be analysed back to a common base document. One possible technique for this is described in [11]. This technique finds all basic edit operations.

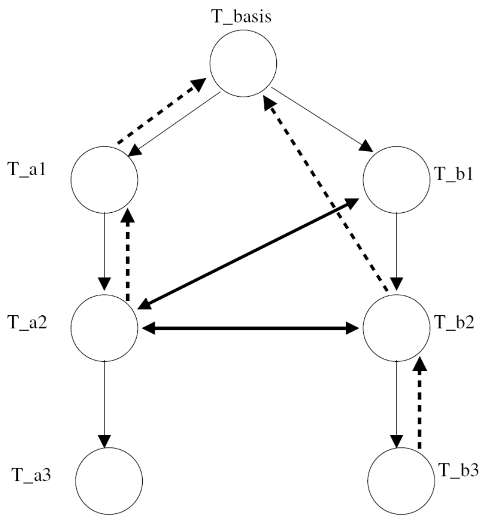


Figure 2. Revision graph with conflicts and dependencies

Fig. 2 shows an example of a revision graph with two branches. Revisions are represented by the nodes of this graph; transactions which lead to a given revision are represented by directed edges with solid lines. T_basis represents the common ancestor of the variants $T\_a1 \dots T\_a3$ and $T\_b1 \dots T\_b3$, respectively. Dashed arrows represent dependencies, double-arrows represent conflicts.

5.2 Conflicts and Dependencies

Edit operations create, change or remove model elements. Hence, each edit operation is executed within a specific context, which consists of the model elements used or affected by the edit operation. Two edit operations in *the same development branch* are defined as *dependent* upon each other if the later one depends upon the effects of the earlier operation, e.g., if the later operation changes or removes model elements that have been created or changed by the earlier one. Two edit operations in *different development branches* are in *conflict* if they affect a common model element, i.e. if the intersection of their contexts is not empty. In general, there is a conflict between two edit operations in different branches, if the sequential execution of both produces different results in the different sequential orders. In addition to these structural conflicts, consistency constraints may be violated by edit operations (including insertions and deletions). An example for such consistency constraints are cardinalities. The effect of two operations may result in a cardinality exceeding upper or lower bounds defined by the meta-model.

Two *transactions* are defined to be dependent or are in conflict, if at least one pair of edit operations contained by the transactions is dependent upon each other or in conflict, respectively.

5.3 History-based Merging

Arranging edit operations into transactions has the potential to enhance the merge process in several aspects. First, from a developer's point of view, transactions represent the changes made to a model on a much higher level of abstraction than edit operations, which are most often too fine-grained, and secondly, they form self-contained changes that may be potentially related with particular requirement specifications. Thus, merge decisions become more comprehensible and are better documented. Taking merge decisions on a more coarse-grained level, i.e. the abstraction level of transactions, also reduces the number of merge decisions that have to be made interactively. If transactions are used as the basis for merge decisions, it is guaranteed that merge results are semantically correct and complete with respect to the originally intended changes. We thus propose a merge process which is based on transactions and which consists of the following main steps:

1. Automatic analysis of the version history data according to [11].

2. Definition of transactions, for example based on the commits made to the repository.

3. Calculation of dependencies and conflicts of edit operations and transactions. Dependencies lead to clusters of transactions for which a consistent merge decision is required [5].

4. Computation of the amount of work included in a transaction. In a first approximation, this value is typically defined as the number of operations included in the transaction. The changes are possibly weighted depending upon the underlying edit data type. Generally, this value can be derived from any difference metrics, which allow us to measure the significance and the amount of change [10].

5. Determination of initial recommendations for merge decisions at the abstraction level of (clusters of) transactions. They can be obtained using different strategies. Sometimes, it is reasonable to give one of the branches a higher priority, which then "wins" all merge conflicts. Sometimes, one wishes to preserve a maximum amount of work and wants to optimize the set of merge decisions (see below).

6. Manual correction of the automatically proposed merge decisions at the abstraction level of transactions. Each manual decision causes a re-calculation of the optimal transaction sequence according to step 5. Typically, this is an iterative process.

7. Manual treatment of discarded transactions. Here, a domain expert can integrate non-conflicting operations into the merge result. This can be achieved through selecting single edit operations or through modifying operation parameters. Finally, a transaction can be marked as "integrated" when edit is finished. It is worth mentioning that, as opposed to the previous steps, the developer is responsible for preserving model consistency.

8. Automatic creation of a merge result based on the merge decisions, which if desired, can be directly committed to the repository.

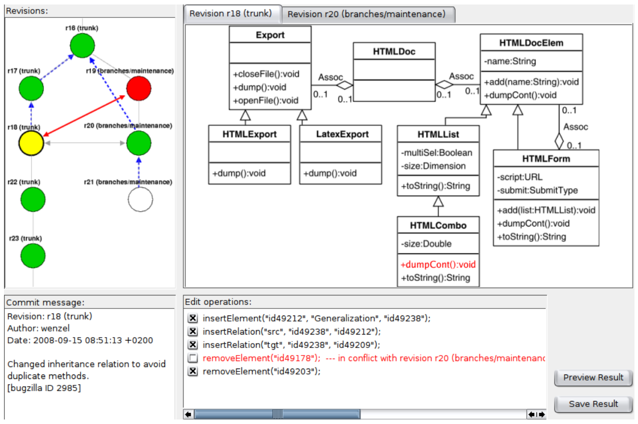
The current state of a merge process, i.e. the selected transactions and edit operations, can easily be saved persistently. Thus, the merge proceedings can be interrupted and be resumed later at any time.

Optimization of Merge Decisions. Merge decisions for transactions are not independent due to possible dependencies between the transactions. As a result of this, one can be faced with a large number of admissible sets of merge decisions. Moreover, one can use several value functions for transactions, e.g. the number of creations of certain element types can be counted separately. In sum, this leads to a multi-dimensional optimization problem, which can lead to several pareto-optimal sets of merge decisions.

However, it is questionable whether the automatic optimization of merge decisions can substantially reduce the work required to manually check the result and the work required in steps 6 and 7.

6 A Prototype of a History-based Merge Tool

A proposal for a merge tool which supports history-based merging is depicted in Fig. 3. The left hand side of the graphical user interface shows the revision graph, including all conflicts and dependencies. Some meta-data related to the currently selected revision are displayed in the window below, as for example the commit message. Transactions that should be included in (or excluded from) the merge result can be selected by means of the revision graph. The rearrangement of the operations contained in transactions into smaller transactions is achieved by means of a customized dialog, which allows a designer to extract a specific subset of operations of a given transaction that is then arranged into a new transaction.



Commit message:
Revision: r18 (trunk)
Author: wenzel
Date: 2008-09-15 08:51:13 +0200

Changed inheritance relation to avoid duplicate methods.
[bugzilla ID 2985]

Edit operations:
```
☒ insertElement("id49212", "Generalization", "id49238");
☒ insertRelation("src", "id49238", "id49212");
☒ insertRelation("tgt", "id49238", "id49209");
☐ removeElement("id49178");  --- in conflict with revision r20 (branches/maintenanc
☒ removeElement("id49203");
```

Figure 3. A History-based Merge Tool

On the right hand side of the graphical user interface, single revisions of the model as well as the preliminary merge result can be examined. In the top right view, a model is shown in usual notation. The bottom right view lists the atomic edit operations of a transaction. Moreover, this view allows a designer to split transactions. Here, splitting includes the possiblity to decide whether only a portion of the edit operations contained in a transaction should be executed and thus be included in the merge result. Further, new operations can be inserted manually into existing transactions.

In the selected example presented in Fig. 3, the deletion of the operation *dumpCont()* has been excluded from the transaction of revision r18. Thus, the operation is preserved. The conflict with revision r20, which includes the operation *dumpCont()* in its required context, is solved.

7 Conclusion

The merging of models is insufficiently supported by contemporary modelling tools. Many principles and subtleties that have emerged and proved successful in merging textual documents cannot be adopted for the merging of models. In this paper, we have analysed the fundamental problems encountered in merging models. Based on that analysis, we have presented a new approach for merging models that is advantageous in four respects. Firstly, while most conventional approaches merge models based on their current states, our approach emphasizes the edit operations that have lead to a given state of a model, based on the version history. This significantly reduces the number of merge decisions. Secondly, our approach allows designers to merge different revisions based on change definitions at a higher level of abstraction, i.e., transactions. Due to the transactions, the model remains in a consistent state. Thirdly, we refer to an edit data type which preserves model consistency with respect to its model type. Thus, the required syntactical and semantic correctness of models cannot be undermined by merge processes. Fourthly, the granularity of the model parts to be merged ranges from entire transactions to single edit operations as the atomic elements of transactions, which ensures the flexibility of our approach.

References

[1] Bendix, L.; Emanuelsson, P.: Diff And Merge Support For Feature Oriented Development; p.31-34 in: Proc. CVSM08, Leipzig ; ACM; 2008

[2] IBM: Rational Software Architect `http://www.ibm.com/software/awdtools/architect/swarchitect/`, 2008

[3] Kelter, U.; Schmidt, M.; Wenzel, S.: Architekturen von Differenzwerkzeugen für Modelle; p.155-168 in: Software Engineering 2008; LNI 121, GI; 2008

[4] Kelter, U.; Wehren, J.; Niere, J.: A Generic Difference Algorithm for UML Models; Proc. GI-Fachtagung Software Engineering 2005, Essen, LNI; 2005

[5] Lippe, Ernst; van Oosterom, Norbert: Operation-based Merging; p.78-87 in: Proc. Fifth ACM SIGSOFT Symp. Software Development Environments; ACM SIGSOFT SW Eng. Notes 17:5; 1992

[6] Matlab/Simulink Software Products `http://www.mathworks.de/products/simulink`, 2007

[7] Mens, T.: A State-of-the-Art Survey on Software Merging; In: IEEE Transactions on Software Engineering 28:5; 2002

[8] Murta, L.; Correa, Chessman; P., Joao G.; Werner, C.: Towards Odyssey-VCS 2: Improvements Over A UML-based Version Control System; p.25-30 in: Proc. CVSM08, Leipzig; ACM; 2008

[9] Treude, S.; Berlik, S.; Wenzel, S.; Kelter, U.; Difference computation of large models; p.295-304 in: Proc. ESEC/FSE 2007, ACM; 2008

[10] Wenzel, S.: Scalable Visualization of Model Differences p.41-46 in: Proc. CVSM08 , Leipzig; ACM; 2008

[11] Wenzel, S.; Hutter, H.; Kelter, U.: Tracing model elements; in: Proc. of the 23rd International Conference on Software Maintenance (ISCM'07), Paris, France; 2007