Computer Supported Cooperative Work
DOI 10.1007/s10606-012-9172-4

Turning Conflicts into Collaboration

Konrad Wieland, Philip Langer, Martina Seidl, Manuel Wimmer & Gerti Kappel
Vienna University of Technology, Business Informatics Group, Vienna, Austria (E-mail: wieland@big.tuwien.ac.at; E-mail: langer@big.tuwien.ac.at; E-mail: seidl@big.tuwien.ac.at; E-mail: wimmer@big.tuwien.ac.at; E-mail: kappel@big.tuwien.ac.at)

Abstract. In model-driven software development, software models are the main artifacts used not only for supporting brainstorming, analysis, and design purposes, but also for generating executable code. Such software models are usually not created by one single developer, but within a team. To coordinate team work, versioning systems have proven to be indispensable for managing modifications performed by different modelers at the same time. When concurrently performed modifications are contradicting each other, the standard versioning paradigm requires the person who detected the conflict to resolve it immediately in order to keep the evolved artifacts in a consistent state. Whereas this approach works well in later phases of the software development process, in early phases, when the development team had not established a consolidated view on the system under development yet, the conflicts might provide valuable information on the various intentions of the modelers. This information might be lost if removed in an undocumented manner by a single modeler. We propose an alternative versioning paradigm for models, where conflicts are temporarily tolerated and discuss its technical realization for current modeling languages such as the UML. The resolution of conflicts is then not performed by one single modeler but within a team so that a consolidated version of the model is obtained.

Key words: model-driven engineering, model versioning, collaborative modeling

1. Introduction

Model-driven engineering (MDE) recently gained high momentum in software development, changing the way how modern systems are build. In MDE, the task of programming, i.e., writing code in a typically textual programming language such as Java, is replaced by modeling in a language such as the Unified Modeling Language (UML). Then the powerful abstraction mechanisms of models are no longer only used for documentation purposes, but also for compiling executable code directly out of models (Bézivin 2005).

With the rise of MDE, several problems solved for traditional software engineering became evident again because their well established solutions are not directly transferable from code to models. For example, standard techniques for

Published online: 19 September 2012

handling software evolution, such as versioning, perform poorly if directly used for models. From a technical point of view, these incompatibilities might be explained by the graph-based structure of models, which might be taken into account by dedicated algorithms for matching, comparing, and merging models. Furthermore, models offer a higher level of abstraction than code by distributing various details over different views. In UML for example, the static structure is shown in a Class Diagram, the dynamic behavior can be specified by an Activity Diagram. In contrast to code, which is a translation of the specification into executable instruction usually written by human programmers, models are much closer to the specification and are refined iteratively until they contain enough information to generate code automatically.

Modeling activities always capture the modelers' personal view on the system under development. Consequently, the fact that each team member perceives the world or the system differently is directly reflected by the models. Especially in early phases of the software development life-cycle, when the team did not establish a consolidated view on the system yet, the understanding and the intention of the different team members might diverge.

One challenge within the modeling process is to create a common knowledge base. Tool support is needed allowing the creation of a common representation of subjective knowledge within the team in order to avoid misunderstandings. In this situation, awareness about conflicting modeling activities might be valuable in order to find a common solution by learning about the reasons that lead to the conflicts (Renger et al. 2008).

The traditional tool support for collaborative software development usually follows the paradigm of either avoiding conflicts or of resolving conflicts as soon as possible. Conflict avoidance is for example realized by the means of *pessimistic version control systems* which lock artifacts for exclusive modification by exactly one developer. In contrast, *optimistic versioning approaches* support distributed, parallel team-work. This comes along with the price of conflict resolution when concurrently evolved versions of one model are merged. Figure 1 (a) shows a typical versioning scenario with conflicting modifications. The modeler Alice creates a new version of a model (V1) and commits this model to the central model repository. The modelers Harry and Sally check out the current version and perform their changes in parallel. Harry deletes an element which is extended by Sally at the same time. Assume that Harry is the first who finishes his work and he is also the first who checks in his version into the repository. Afterwards, Sally tries the same, but the VCS rejects her version V1b, because her changes are conflicting with Harry's changes. In a standard versioning process, Sally is responsible for resolving the conflict immediately. Assuming that Sally does not want to corrupt Harry's work, still information might get lost, because she might not be aware of all intentions Harry had when doing his work. Also Harry is not aware how Sally is doing the merge. Finally, one conflict-free version is checked in with the consequence that some modifications may be

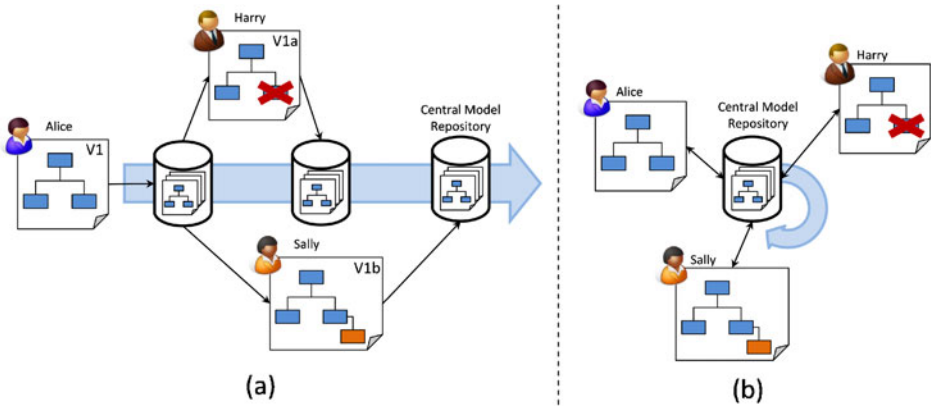

Figure 1. (**a**) Continuous Conflict Resolution and (**b**) Conflict Tolerance.

inevitably lost as they are removed in an undocumented manner. Such an approach is adequate for code when the specification of the system is already established (e.g., in terms of test cases) and the code has to be executable at a certain point in time. In contrast to code, models are often used in an informal manner for collecting ideas and discussing design alternatives in brainstorming periods. Models may serve as sketch in early project phases (Fowler 2003) and are used to manage and improve communication among the team members by establishing common domain knowledge. In such a scenario, this loss of information is a particular problem. The discussion of the conflicts might had helped to eliminate flaws in the design which are probably harder to eliminate at the later point in time of the software development life cycle (Brosch et al. 2009).

In this context, a paradigm shift as depicted in Figure 1(b) is the solution. In many situations, it is desirable to keep all (or at least many) of the modifications, even if they are conflicting. The conflicts may help to develop a common understanding of the requirements on the new system. To support versioning in early project phases, we propose a versioning system which temporarily tolerates conflicts enabling a creative design process without destroying the model's structure in order to still use available modeling tools for editing.

1.1. Contribution and outline

This paper extends our work (Brosch et al. 2010b) by an elicitation of user requirements on model versioning systems established by the means of expert interviews. Furthermore, we also considered additional kinds of conflicts and introduced well-defined conflict detection patterns, as well as merge rules. Also a conflict resolution model and conflict life-cycle has been elaborated to obtain a consolidated version of the model. Furthermore, we broadened the scope of the related work and provide insights on our prototypical implementation. Finally, a

quasi-experimental study was conducted to evaluate our approach and the results are presented and discussed within this paper.

The outline of the paper is as follows. In the next section, we present the methodology we used to develop the proposed approach. In Section 3, we discuss the related work of different research areas. Then, we discuss the requirements arising from practice, and continue with a running example in Section 5 which severs as a show case in the following sections. In Section 6, we present the different kinds of conflicts that may occur in model versioning and how they can be represented. Section 7 introduces our approach of tolerating conflicts when merging different versions of a model. The consolidation phase is discussed in Section 8 followed by a description of the implementation in Section 9. The results of the evaluation are presented in Section 10 and, finally, we conclude with a critical discussion and future work in Section 11.

2. Methodology

The methodological approach used in this paper conforms to the design science approach presented by Hevner et al. (2004). In general, the types of output produced by design research are representational constructs, models, methods, or instantiations (March and Smith 1995). The principle layout of design science research and how it is reflected in this paper is depicted in Figure 2.

The development of an artifact and its refinement based on different evaluation methods constitute the core of this research process. The design is driven by the environment defining business needs and addressing these needs assures research relevance. Existing foundations and methodologies are applied in the design of the artifact and, after evaluating this artifact, the knowledge base is updated or

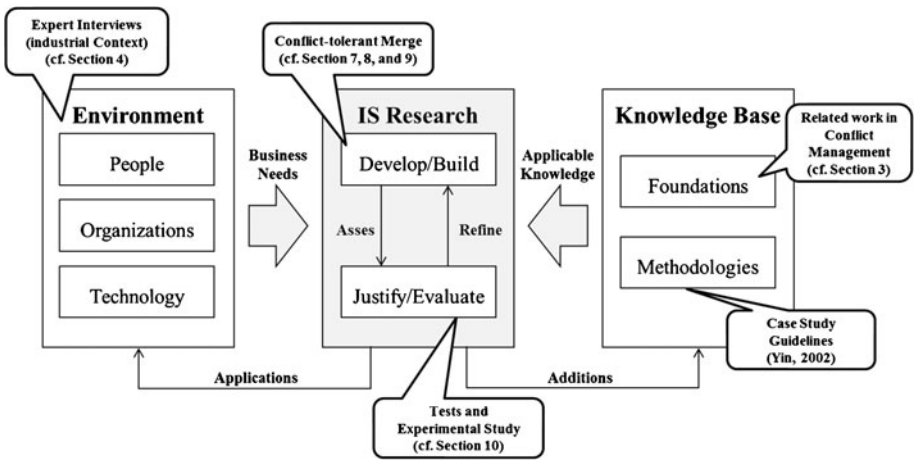

Figure 2. Information Systems Research Framework; Adopted from Hevner et al. (2004).

new knowledge is added. In the following, we outline how we use this approach within this paper.

The contributions of this paper produce a viable artifact in terms of a new paradigm for versioning software models, namely the *Conflict-tolerant Merge*. This approach was developed to support relevant business needs, which were elicited by conducting several expert interviews (cf. Section 4). During the development phase, tests were conducted for refining and improving the individual artifacts. In addition, the utility of the artifacts are rigorously demonstrated via an experimental study (cf. Section 10). The experiment was conducted on the basis of the general applicable case study guidelines presented by Yin (2002). In addition, our knowledge base also provides foundation of different research areas, such as "versioning" or "inconsistencies in software engineering", which build related work to our contributions presented in this paper (cf. Section 3).

3. Related work

Concerning the goal of the paper, we distinguish between two threads of related work. First, we elaborate on approaches and tools that are coping with conflicts occurring in collaborative software engineering processes in general. Second, we survey approaches focusing on conflict visualization and annotation.

3.1. Coping with conflicts

Over the years, three different ways to handle conflicts in collaborative software development have solidified (Edwards 1997; Mens 2002): (i) avoiding conflicts, (ii) resolving conflicts, and (iii) tolerating conflicts.

Conflict avoidance Probably the simplest way to deal with conflicts is to establish mechanisms which make the occurrence of conflicts impossible. Two prominent representatives of collaborative tool support, which hinders conflicts to emerge, are real-time collaborative editing systems and pessimistic versioning systems. Conflicts cannot emerge in the former, because the developers share one screen, such that they synchronously work together on the same artifact. Whenever one developer touches one element, it is immediately highlighted with the consequence that, at any time, every involved developer is aware of the modifications the others are performing. Conflicts cannot occur in the latter, because exclusive rights to modify an artifact are granted.

Already in 1968, Engelbart and English showcased an approach to collaborative editing in the conferencing system presented at his "mother of all demonstrations" (Engelbart and English 1968). Over the years, several dedicated environments for real-time collaboration have been proposed (for textual artifacts as well as for models) which provide sophisticated notification and communica-

tion mechanisms indicating that a resource is currently touched by an other team member (e.g., Shen (2003); Fitzpatrick et al. (2006)). Examples of collaborative modeling editors are SLIM (Thum et al. 2009), DAWN (Fluegge 2009), and SpacEclipse (Gallardo et al. 2011).

Pessimistic versioning, also known as locking, is supported by most standard versioning systems such as Subversion (SVN), as well as by several dedicated model versioning systems, allowing only the developer to modify a certain artifact if she possesses the lock on this artifact. Code versioning (pessimistic as well as optimistic which we discuss below) has a long history in computer science—the first systems date back to the 70s—and have become indispensable for efficient software development (Conradi and Westfechtel 1998; Mens 2002). As discussed before, a particular challenge in model versioning is posed by the selection of an adequate level of granularity determining the size of a lock since this has significant impacts on the work efficiency. The price of conflict avoidance, resulting in always consolidated states of the system, has to be paid with restricted flexibility during work, because the developers have to coordinate meetings for the real-time editing approach or they have to spend time idle when waiting for resources to be unlocked again. According to Estublier et al. (2005) and our survey presented in Wieland et al. (2011) pessimistic versioning restricts the amount of parallel work and typically leads to unwanted delays within a project.

Conflict resolution When applying optimistic versioning, the developers intentionally risk conflicts for more flexibility during work, because optimistic versioning tools put no dependency constraints on the developers. From time to time, when they save their work in the central repository, the different versions of the artifact under development have to be merged. If a conflict is detected, it has to be resolved immediately; otherwise it may not be committed in the central repository. Thus, the developers may assume that the most recent version stored in the central repository never contains any inconsistencies, i.e., that none were introduced when merging the concurrently evolved versions. The resolution step may be supported with the help of specific rules or policies (Munson and Dewan 1994). Although very simple, line-wise comparison has proven to be effective for textual artifacts such as source code. For software models, more sophisticated comparison and conflict detection algorithms are required. This is because they have to take into account the graph-based structure and their rich semantics which would be lost if models are serialized in flat text files (Altmanninger et al. 2009b). Therefore, recently several dedicated model versioning approaches emerged (cf. Altmanninger et al. 2009a, for a survey). In Taentzer et al. (2010) and Ehrig et al. (2011), theoretical foundations are presented how model versioning can be formalized on the basis of graph modifications. Despite the precise conflict detection components, merge problems might still occur, because conflicts are

usually resolved by the person who does the later check-in and this person might be (on purpose or by accident) undo the modifications performed by other team members. For this problem, we offered a solution in Brosch et al. (2009), with which we suggested to perform conflict resolution in the team instead of by an individual in order to avoid misunderstandings. In many situations, conflicts indicate misunderstandings in the specification, which might be especially valuable in early phases of the development process. Hence, the idea is to collect and to tolerate conflicts and resolve them at a later point in time that is discussed in the following.

Tolerating conflicts During the late 80s to the late 90s, several works have been published which aim at managing inconsistencies. Inconsistencies may arise, when syntactic properties (context-free or context-sensitive) or semantical concerns (structural or behavioral) are violated. Conflicts may be seen as a certain kind of inconsistencies in the software engineering process. One of the most interesting commonalities of these works is that the authors considered inconsistencies not only as negative result of collaborative development, but also see them as necessary means for identifying aspects of systems which need further analysis or which need to reflect different viewpoints of different stakeholders (Nuseibeh et al. 2001; Spanoudakis and Zisman 2001). Originally, the need for inconsistency-aware software engineering emerged in the field of programming languages, especially when very large systems are developed by a team. Schwanke and Kaiser (1988) have been one of the first who proposed to live with inconsistencies by using a specially adapted programming environment for identifying, tracking, and tolerating inconsistencies to a certain extent. A similar idea was followed by Balzer (1989) for tolerating inconsistencies by relaxing consistency constraints. Instead of forcing the developer to resolve the inconsistencies immediately as they appear, he proposed to annotate them with so called "pollution markers". Those markers comprise also meta-information for the resolution such as who is likely capable to resolve the inconsistency and for marking code segments which are influenced by the detected inconsistencies. Furthermore, Finkelstein et al. (1994) presented the "ViewPoints" framework for multi-perspective development allowing inconsistencies between different perspectives and their management by employing a logic-based approach which allows reasoning even in cases where inconsistencies occur (Hunter and Nuseibeh 1998).

Conflict-tolerant merging is to a certain extent also reflected in current text-based versioning systems such as SVN or the approach of Lukosch and Leisen (2009) focusing on wiki pages. For instance, if a conflict occurs using SVN, a merged version is generated, which duplicates the text lines that have been concurrently modified. Moreover, these duplicated text lines are surrounded by dedicated conflict markers to indicate the conflict more prominently. Developers may now use this version, mark it as resolved and save them into the repository

again. In this way, the resolution of the conflict is postponed to another point in time. However, in contrast to the approach proposed in this paper, the conflicting changes are not really merged; the concurrently changed lines are simply duplicated and important meta-data about the conflict is missing. Moreover, the conflict annotation is intermingled with the code and destroys readability and understandability of both the code and the conflict.

The presented approach of this paper is in line with the mentioned approaches for tolerating inconsistencies in software engineering. In particular, we also aim at detecting, marking, and managing inconsistencies. Concerning the marking of conflicts, we also have a kind of pollution markers as introduced by Balzer (1989). However, we are strongly focusing on the parallel development of models, thus we have additional kinds of conflicts as discussed in Section 6. Furthermore, we are not only marking, but we have also to merge a tailored version, in which it is possible to mark the conflicts and inconsistencies. Our goal is to support this approach without adapting the implementation of the modeling environment and versioning system. This is mainly supported by the powerful dynamic extension mechanism of EMF Profiles (cf. Section 9). A dedicated conflict profile is used to annotate and visualize conflicts within the model without polluting it.

3.2. Visualization and annotation of conflicts

In the following, we review approaches which support the visualization of changes and conflicts. Therefore, we consider not only the literature on software modeling, but also the literature on ontology engineering. Ontologies are usually established by communities, and therefore sophisticated mechanisms are required to enable collaborative work. As the ontology engineering process is highly interactive, mechanisms are required to keep conflicts in order to establish a consolidated opinion of the community.

Software modeling Mehra et al. (2005) and Ohst et al. (2003) proposed approaches for the visualization of differences between model versions by using different coloring and highlighting techniques. The modifications are shown in *unified diagrams* which incorporate the changes of both users. The approach of Ohst et al. has been implemented in Niere (2004), but due to a restricted merging approach only conflicts concerning contradicting updates of attribute values as well as element moves are marked explicitly in the unified diagrams. Furthermore, tool-specific extensions have to be implemented for modeling editors in order to use this approach. Mehra et al. also report on changes concerning the concrete syntax. For each movement of the shape of a model element, the original as well as the new position of the shape connected by a line is shown. Many overlapping highlighted model elements are generated when a large number of changes has occurred. The approach has been implemented for the meta-CASE tool Pounamu for providing generic visualization support for

modeling languages defined in Pounamu, but for, e.g., UML modeling environments, there is no support available.

Ontology development Ontologies are kind of conceptual models. Thus, there are many similarities between building an ontology and a structural model such as an UML class diagram. Furthermore, several works are geared towards integrating models and ontologies such as Kappel et al. (2006). Ontologies cover common knowledge of a certain domain, and usually the building of an ontology is a community activity in order to collect common domain knowledge and in order to establish a common terminology. Hence, the ontology engineering community needs tools for collaboratively developing ontologies (Sebastian et al. 2008). Often, ontologies are constructed by the means of Wikis such as LexWiki[1] intended to define terminologies. Therefore, the participating engineers can comment the current status and propose changes in a text-based manner by annotations. These annotations are later examined by curators which are editing the ontologies in standard ontology editors separated from LexWiki. OntoWiki (Auer et al. 2006) supports to change and rate ontology definitions via a Web-based interface. However, OntoWiki cannot represent conflicting changes explicitly. Collaborative Protégé (Tudorache et al. 2008) also allows for collaborative ontology development with annotations on ontology changes, proposals, votings, as well as discussions. The authors recognize the need for synchronous and asynchronous development as one of the main requirements for ontology engineering, but for the moment only synchronous development is supported. Consequently, conflict detection and visualization is not treated by these approaches in contrast to this paper. Furthermore, ontologies are developed in abstract syntax using tree editor, thus no concrete syntax conflicts are considered.

4. Requirement elicitation

In 2010, we have interviewed ten experts in the area of (model-driven) software development. Our interview partners gave us deep insights in their development processes and versioning habits. Although they came from very different domains, they have one in common: in their companies, they develop software collaboratively. These interviews gave us a profound basis for eliciting requirements, which arise from practice. In this paper we provide an excerpt of these interviews. For more details the interested reader is kindly referred to Wieland et al. (2011).

To understand the requirements from the perspective of software development practices, we decided to take a qualitative approach and conduct semi-structured narrative interviews (Wengraf 2001). In preparation, we identified a number of broad themes that we hoped to cover with the participants and wrote these up as an interview guide. These included: a discussion of their role, a presentation of their

[1] http://biomedgt.org

projects, stories of their collaborative work, reviews of their versioning habits, etc. The actual interviews were then conducted as open-ended conversations, enabling us to follow leads as brought up by the interviewees, with the guide playing a background shaping role to how we engaged in the conversation. The duration of the interviews varied between half an hour and two hours and took place usually in the offices of the interview partners. The interviews were audio recorded and later transcribed. Since most of the interviews were conducted in German, the quotes presented below were translated to English as exactly as possible. A qualitative thematic analysis was conducted with the transcripts to draw out emergent themes.

Out of these interviews we have extracted three major requirements, which are presented in the following.

Conflict tolerance When working in parallel and independent of each other like in optimistic versioning systems, conflicts or constraint violations may occur, when the same artifacts are concurrently modified. Developers try to avoid conflicts, because the conflict resolution process is usually a time-consuming and error-prone task. In standard VCSs, conflicts have to be resolved immediately by the developer who is committing her changes. In the interviews it was often mentioned, that committing the changes to the central repository should be possible without worrying about conflicts. For example, one developer states:

> "I want to commit my changes to the repository, but I don't want to resolve conflicts immediately. […] For me, it is often not possible to resolve a conflict alone."

Since avoiding conflicts is not always possible and conflict resolution may lead to undesired results, a conflict tolerant approach is needed that tolerates conflicts to resolve them later on in a collaborative setting. All changes of all participants should be incorporated in a unified model, conflicts should be marked to resolve them later on.

Collaborative consolidation In traditional VCSs, the developer who commits her changes bears the full responsibility when resolving conflicts, which may lead to undesired results. The developer does not always fully grasp the intentions behind the changesof the other developers. Thus, it is very difficult or impossible to finally have a consolidated version of the model reflecting the intentions of all developers. One developer states:

> "Basically we try to avoid conflicts. But, when conflicts arise, we have a big problem. […] The conflicts have to be resolved collaboratively in a face-to-face session. They build a good basis for discussion."

In the interviews, it is often mentioned, that conflict resolution has to be done by the responsible developers, not by a single person, who has checked in her

changes. Conflicts should not be seen as negative results of collaboration, but as good basis for discussing different view points.

Comprehensible evolution When different developers work in parallel and independently, it might easily happen, that the modification are not comprehensible any more. In traditional VCSs it is nearly impossible to get to know in which way a conflict is resolved, because this provenance information is missing. One interviewee for example claimed:

> "For me, as senior developer, it is often very important, to see which part *(Note: of the model)* is changed by which developer and how it is changed. […] When it comes to a conflict I want to know later on, how this conflict was resolved and who was responsible for the resolution decision."

This information about the resolution process should be explicitly available in the versioning system. The evolution of a system is getting more comprehensible for the participants when raising the awareness of changes, conflicts as well as the resolution process.

5. Running example

The example depicted in Figure 3 describes a merge scenario which demonstrates typical problems when developing models in a distributed team following the standard versioning process. All team members may check-out and modify the model in parallel and independent of each other without any restrictions. And

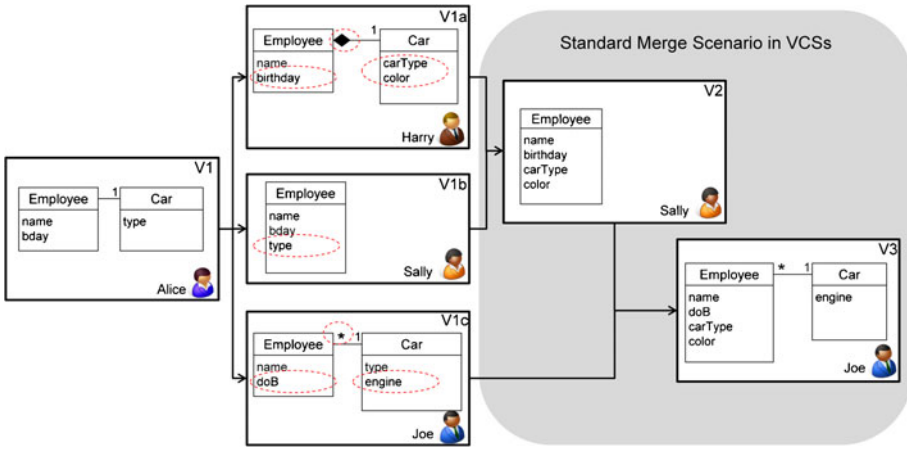

Figure 3. Running Example: Merging in Current VCSs.

when a conflict arises, the developer who is committing her changes is solely responsible to resolve the conflict immediately.

V1 Alice creates a new model in terms of a UML Class Diagram[2] containing the two classes Employee and Car. She adds the attributes name and bday—representing the birthday of a person—to the class Employee and type to the class Car. Finally, she defines an association between these two classes and sets the multiplicity to 1 to define that one employee is assigned to exactly one car. After she has finished, she checks in the new model (called V1) into the central repository of the VCS. Now, the modelers Harry, Sally, and Joe want to continue working on this model and therefore, they check out the current version V1 of the model from the central repository to perform their changes.

V1a In Harry's opinion, a car is always owned by exactly one employee. Therefore, he modifies the association to express a containment relationship (noted by the black diamond). In addition, he changes the attribute bday of the class Employee to birthday and adds a new attribute, namely color, to the class Car. In addition, he changes the name of the attribute type to carType.

Resolution Because Harry is the first to check in, he has no conflicts to resolve. His version is now the latest version within the repository.

V1b Since in Sally's opinion one car always belongs to one employee, she inlines the class Car in order to ensure efficient querying. In particular, she moves the attribute type from class Car to Employee and subsequently deletes the class Car.

When Sally commits her changes, the VCS rejects her modifications because they are conflicting with the already performed changes of Harry. A so called "Delete/Update conflict" occurs since the deleted class Car has been changed by another modeler (Harry). Consequently, Sally is responsible to resolve this conflict according to the process of standard VCS.

Resolution In Sally's opinion, the class Car is still unnecessary in the model, although Harry has updated this class by adding a new attribute. Therefore, she decides to inline the added attribute of Harry, namely color, into the class Employee and to still abandon the class Car. Version V2 now consists of the class Employee containing the attributes added or updated by Alice, Harry, and Sally.

V1c Joe has also checked out the version of Alice (V1) and performs his changes in parallel to Harry and Sally. He renames the attribute bday to doB (*d*ate *o*f *b*irth) in the class Employee and adds to the class Car a new attribute called engine.

[2] http://www.omg.org/spec/UML/2.2

Furthermore, Joe is of the opinion that one car may belong to several employees and therefore, he sets the multiplicity of the association to unbound indicated by the asterisk in the model.

Resolution Now, Joe tries to check in his version V1c into the central repository but different conflicts are reported by the VCS, because Joe's version is now conflicting with the current version V2 in the repository. Recall that V2 is the version of the model covering Harry's and Sally's modifications consolidated by Sally. Now an "Update/Update conflict" is reported because both, Harry and Joe, have updated the attribute bday in different ways. Joe decides to take his version namely doB. In addition, also a "Delete/Update" conflict is reported, because Joe has updated the class Car by adding a new attribute—namely engine—to this class. He decides not to delete the class Car in contrast to the decision of Sally.

The new version V3 of the model is put into the repository after Joe has resolved the conflicts. Although version V3 is a valid model, it contains several severe flaws resulting from the conflict resolutions. The final version of the example model does not reflect all intentions of the participating modelers, because only one single modeler was responsible for the merge of two versions. Also when assuming that all modelers do their best to resolve occurred conflicts, it is particularly difficult for them to understand the motivation behind the changes of the others, especially, if more than two modelers performed changes in parallel.

In particular, the version V3 of the example described above does not reflect the idea of Harry to consider a car as *part of* one employee as expressed by the composition. It does also not reflect the idea of Sally to have only one single class for efficiency reasons. Furthermore, in V3 the attributes carType and color have become part of the class Employee that is obviously undesired assuming that the class Car is retained. Over the evolution of the model the information that these attributes are referring to the car get lost since no indications for that exist anymore.

To summarize, the final version of the model contains several flaws and does not reflect several of the modelers' original intentions. The reason for such undesired situation is mainly that only one modeler was responsible to resolve all conflicts on her/his own. Furthermore, currently only limited tool support is available for such tasks. As an ultimate result, changes, opinions, and potentials for obviously important discussion are destroyed by this versioning paradigm.

6. Conflicts in model versioning

In the illustrative example discussed in the previous section, various conflicts show the malicious side-effects occurring when conflicts have to be immediately resolved by only one modeler. In order realize a versioning system which allows

deferring the resolution of conflicts until a consensus has been—in the best case—collaboratively worked out, we first have to discuss the notion of conflict in detail in order to obtain a suitable technical representation which results in concrete processable instances of conflicts.

6.1. Preliminaries: atomic changes

Conflicts are always caused by parallel changes which are incompatible when they are applied one model at the same time. Thus, to detect conflicts, the applied changes have to be considered. In the following, we use the terms operation, change, and modification synonymously.

Independent of any modeling language, atomic changes may be detected. These changes are *insert*, *update*, *delete*, *use*, and *move*. With these five operations at hand, any complex modification is expressible. Whereas *insert* adds a new element to the model, *delete* removes an element from the model, and *update* modifies a certain property of a model element. For example, with the update operation the name of a class in a UML Class Diagram can be changed. If a reference from one model element to another element is set, a *use* operation is performed. As a special case of update, the *move* operation has to be emphasized, which changes the container of a model element. This operation is of paramount importance for models, because models are often hierarchically organized, e.g., when we consider UML Class Diagrams, an attribute is contained in a class, a class is contained in a package, a package may be contained again in a package and so on.

6.2. Kinds of conflicts

As we have seen in the previous section, when merging differently evolved versions of one model, various kinds of conflicts might occur. These kinds of conflicts are shortly discussed in the following. For a more detailed categorization of conflicts in model versioning the interested reader is referred to Altmanninger et al. (2009b).

When two changes interfere with each other, i.e., either result in different models when applied in different orders or one change hinders to apply the other change, then conflicts due to overlapping changes shall be reported. Table 1 shows, which change combination leads to a conflict, if they are performed on the same model element in parallel. Conflicts are marked with $\times$, and the combination of non-conflicting changes, which might potentially result in an undesired situation, of which the users should be notified, are marked with $\sim$. In the following, we shortly describe the table and in the next section a detailed presentation of the conflicting situations is presented. One important characteristic of overlapping changes is that no unique and complete merged version can be produced without defining priorities for changes.

Table 1. Overlapping changes.

Insert	Delete	Update	Use	Move	
					Insert
		×	×	×	Delete
		×	∼	∼	Update
				∼	Use
				×	Move

× ··· *Conflict*
∼ ··· *Warning*

Inserts are never considered as source of conflicts in the scope of this paper, because when an element is introduced by one modeler, the other modeler cannot apply contradicting deletion, update, and move changes on the same element. However, some conflict detection approaches report insert/insert conflicts, when the "same" model element is introduced by multiple modelers. This is usually the case, when the comparison is not based on artificial identifiers, but on matching heuristics such as string comparisons. In this paper, we do not consider such situations, because we use only artificial identifiers and a merged model comprising both model elements may be produced anyway.

When two modelers delete the same model element in parallel, no conflict has to be reported, as the intention as well as the result of both changes is the same. All other change combinations applied to the same model element lead to conflicts, e.g., when two modelers update the same feature (such as the name) of a model element, in a different way. In the running example (cf. Figure 3), several Update/Update conflicts caused by the parallel modifications of one element, such as of the attribute bday in the class Employee, occur. In such situations, it is not possible to produce one unique merged version including both changes without extending the modeling language. The same is true for Delete/Update conflicts, e.g., consider our running example where Sally deletes the class Car which is modified by Harry and Joe.

If a model element is deleted and, concurrently, a reference value using the same element as target is inserted, a Delete/Use conflict is reported. Consider a model of two class A and B. One modeler deletes the class B whereas the other modeler concurrently set a reference to the same class. The deleted class B is used as a new reference value and, thus, a Delete/Use conflict occurs.

A special kind of Delete/Update conflicts are Delete/Move conflicts where the update operation is actually a move operation. Of course, when one modeler

moves a model element and the other modeler deletes it, a conflict has to be reported. Finally, another source for conflicts is when one element is moved to different containers by two modelers in parallel. For instance, assume that an attribute is moved from class A to class B by modeler 1 and modeler 2 moved the attribute from class A to class C. The container has to be unique for each model element, therefore this situation must result in a move/move conflict. In fact, a move operation is a special kind of update operation, but drawing the distinction between these operations give the user more detailed information.

Another special combination of changes is when an element is moved to a different container and exactly this element is updated in parallel by another modeler. When both versions are merged, no conflict is reported by the system since both operations can be performed independently of each other. For instance, an element is moved to the new container and, in parallel, it is updated. The same result is produced when the changes are executed in the reverse order. Now one may conclude that such situations are not problematic. However, the modelers should be aware of this situation, because the new context of the moved element may lead to undesired situations. Although coming from another background, Dey and Abowd (2000) define context as "any information that can be used to characterize the situation of an entity. [..]". Furthermore, they define a system as context-aware "if it uses context to provide relevant information [..] to the user, where relevancy depends on the user's task." We also want to make the modelers aware of the new context, i.e., the new container, of the moved element when it is changed in parallel by another modeler. For instance, consider an example in which we have two classes in a model, namely a class Project and a class Task whereby the latter comprises an attribute duration. Now one modeler is adding a constraint to the attribute duration that the values have to be less than ten hours. In parallel, another modeler is moving the attribute duration to Project, because she thinks that the duration of a complete project is more interesting than single tasks. When these two changes are merged, we would obtain a model in which the duration of projects is limited to ten hours. This example shows that the context of an element is very important to reason about the meaning of an element. Thus, in Table 1, ~ is noted which means that not a conflict is reported but a Move/Update *warning* is raised. In such cases the modeler who updates the element should verify if her update makes sense also in the new context after applying the move.

Similar situations occur, when a use operation is performed and, in parallel, the used element is updated or moved. If an element is used by another element and the used element is updated or moved, both operation can be performed, but we

report an use/update or use/move warning to notify the user that an undesired situation might occur which can be checked by the modeler.

Language-specific conflicts In addition to overlapping changes which result in conflicts, two other kinds of conflicts occur in the context of model versioning.

Syntactic conflicts in a model may be regarded as violation of its metamodel, i.e., the specification of the modeling language. Figure 4(a) shows an example where the modifications of two modelers lead to a so called "inheritance cycle" which violates the metamodel constraint forbidding such cycles. This kind of conflict does not hamper the production of a merged version as it is done by overlapping changes. On the contrary, a merged version is necessary before these conflicts can be detected, because these language constraints are defined on the state of models and not on applied operations.

Semantic conflicts occur if the meaning of the merged model is incorrect. Since the semantics and the correct interpretation of a model is difficult to express in a formal way, the detection of such problems is challenging and usually requires human assistance although domain knowledge as encoded in upper ontologies might be supportive. Figure 4(b) shows a Class Diagram containing the class Bird which has two subclasses Dove and Sea Gull. Both classes provide the method fly (). Then one modeler performs a refactoring and shifts the method fly () into the class Bird. At the same time, another modeler introduces a novel class Penguin which is also of type Bird. When standard merging the modifications of both modelers, we have probably an undesired situation at hand, because a penguin being able to fly is specified in the model which contradicts reality.

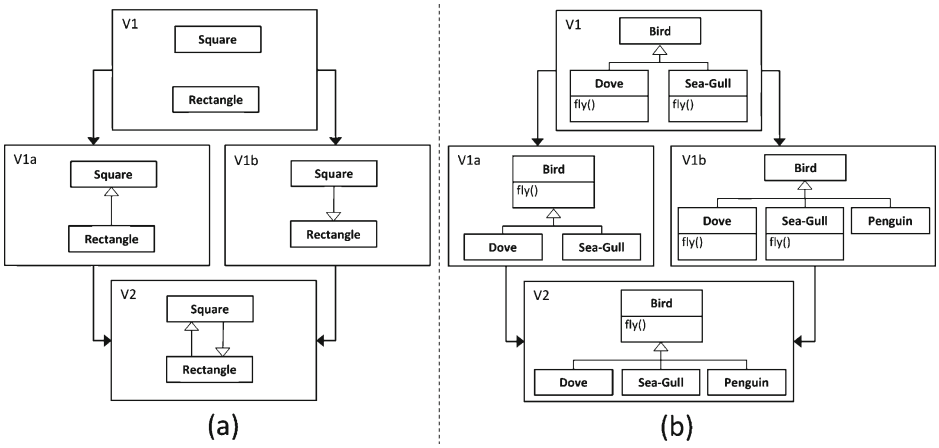

Figure 4. Example Conflicts: (**a**) Syntactic Conflict, (**b**) Semantic Conflict.

Summary In the context of concurrent modeling especially overlapping changes require for additional techniques which allow to represent different viewpoints and alternative designs within one model. Therefore, the next section deals with a dedicated merge approach which allow to integrate also overlapping changes in one model. In addition, this merge allows to build models where subsequently syntactic conflicts may be annotated by reusing model validation frameworks.

7. Conflict-tolerant merging of models

The overall goal of our conflict-tolerant merge is to incorporate all changes concurrently performed by two modelers into a new version. However, in contrast to existing model merging approaches, we address the requirements from practice that developers often wish to avoid being interrupted by conflict resolution; instead they would like to check-in their changes without considering potential conflicts immediately (cf. Section 4). Therefore, the proposed merge algorithm allows for a merged version irrespectively of any occurring conflicts and *without forcing users* to immediately *resolve conflicts*. Of course, the information on occurred conflicts is not lost, but is *annotated* in the model to enable a collaborative resolution at a later point in time (cf. Section 8). To realize the conflict-tolerant merge, we present a set of dedicated merge rules in the following that enable merging overlapping changes (cf. Section 6), that would normally halt the merge, and annotate all relevant information for a later resolution.

7.1. Conflict tolerant merge rules

In the following, we present the different types of conflicts and introduce for each conflict type a dedicated rule specifying how a merged model is produced when two conflicting operations are at hand. The merged model is produced with the help of annotations and priorities for changes. Thus, these two aspects are presented in detail.

In order to allow for generic definitions of the merge rules, we make use of the UML Object Diagram notation to represent the model fragments. This means that each model element is represented by an object. An object has links to other objects as well as attribute values for possessing simple data values. Please note that there is a specific kind of link called containment link which expresses container/containee relationships. In the following figures, such containment links are depicted by a black diamond at the container side. The containment links are especially used in these figures if an element moves, i.e., changes of the container, are involved.

Update/update conflict As depicted in Figure 5, an Update/Update conflict arises if the same feature f1 of object o1 is changed in two different ways in parallel. Since both updates can not be performed when merging without overriding each other, the original value namely x of the feature f1 is preserved in o1.

To mark this conflict, the modified object is annotated with the an UpdateUpdate annotation containing values of both concurrent changes. The annotation includes the involved feature of the object. Finally, meta-information is added to the annotation such as user-related and time-related information. This meta-information is provided for each conflict annotation.

Delete/update conflict In Figure 6 the merge rule for Delete/Update conflicts is depicted. A Delete/Update conflict is reported, when the same object o1 is changed by updating one of its features while it is deleted in parallel.

In this case, we prioritize the update and annotate the object o1 with a Delete/Update annotation. In this annotation, the old value and the update value of the changed feature is preserved. Please note that in order to represent this kind of conflict in a model, the deletion is not executed but only marked by the annotation. The actual deletion is deferred as long as the modelers reach the conclusion that the deletion has a higher priority as the update.

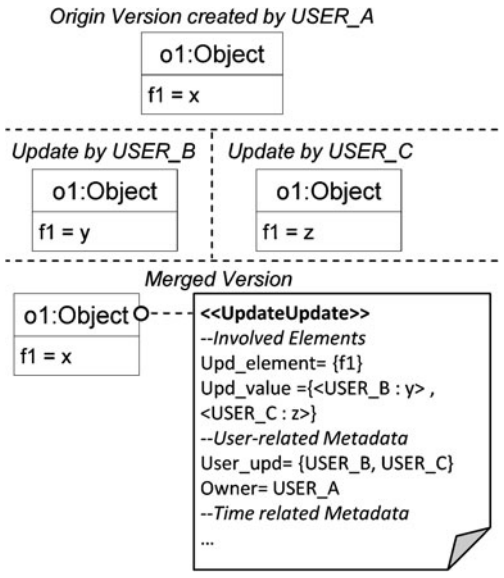

Figure 5. Merge Rule for Update/Update Conflict.

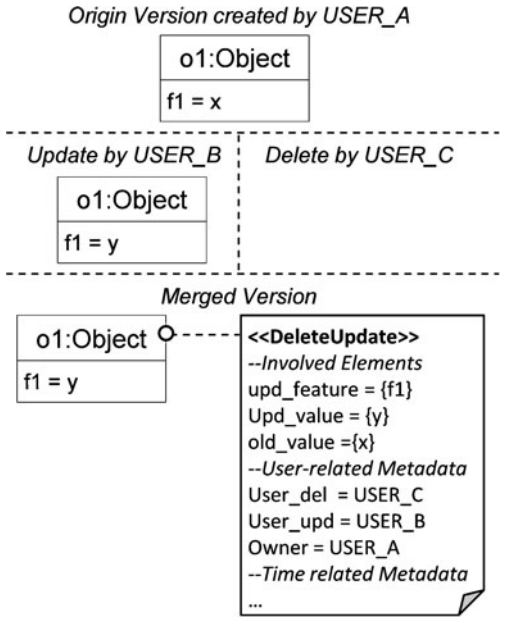

Figure 6. Merge Rule for Delete/Update Conflict.

Delete/use conflict The merge rule for Delete/Use conflicts is depicted in Figure 7. The object o2 is used as reference value of feature f1 of the object o1. In parallel, the used object o2 is deleted.

This conflict is handled in a similar manner as a Delete/Update conflict. The deletion of object o2 is not performed and the reference to this object is still available in the merged version. In this version, the deleted object is annotated and the old value of the changed feature f1 is preserved in this annotation.

Move/move conflict When an object o1 is concurrently moved into different container (c2 and c3) as depicted in Figure 8 a so-called Move/Move conflict arises due to the fact that the container of an object always has to be unique.

This conflict is handled similarly to an Update/Update conflict, i.e., the original structure is retained and the contradicting changes are not executed but only annotated. Since both move operations cannot be executed together, the object o1 is still contained by c1, the original container, after merging and is marked with a Move/Move annotation. This annotation contains both container objects, i.e., c2 and c3, to which object o1 has been concurrently moved.

Delete/move conflict A special kind of Delete/Update conflicts are Delete/Move conflicts. Such conflicts arise when the object, which is moved into another container, is concurrently deleted. Figure 9 illustrates such a situation in

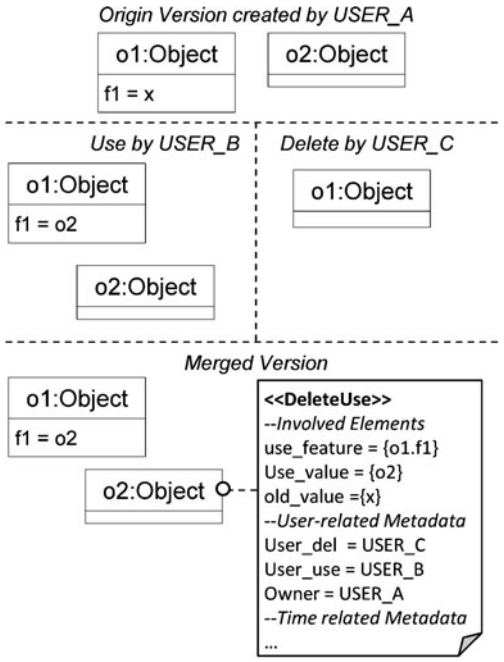

Figure 7. Merge Rule for Delete/Use Conflict.

which the object exttto1 is moved into the container c2 while o1 is concurrently deleted.

As already mentioned, this conflict is similar to the Delete/Update conflict. Therefore it is handled analogously during merging. The delete operation is not executed, the object is moved into the new container, and it is appropriately annotated with a Delete/Move annotation as depicted in Figure 9.

Move/update warning Figure 10 depicts a situation, where the object o1 is moved into a new container c2 and a feature f1 of object o1 is concurrently updated. Basically, both operations can be executed, and therefore no conflict should be reported. However, changing the context of a model element by moving it into another container while having concurrent updates may lead to undesired results.

To give the modelers the chance to review such situations, the moved object is annotated with a dedicated warning to make the modeler aware of a potential problem. In particular, a Move/Update annotation is attached to the moved object which also comprise the old value and the old container of the updated feature.

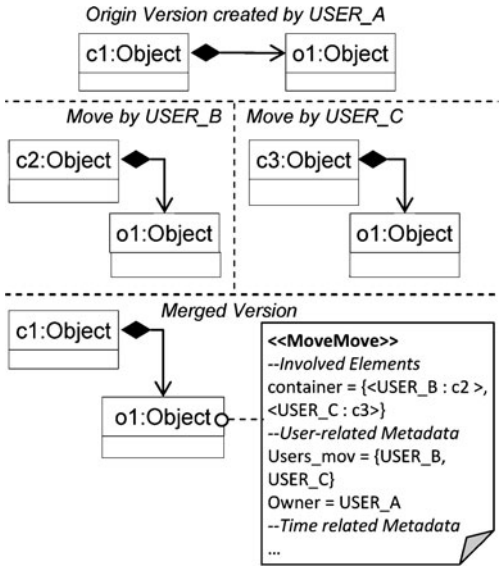

Figure 8. Merge Rule for Move/Move Conflict.

Other kinds of warning are the **Use/Update Warning** and the **Use/Move Warning**. Similar to the Move/Update warning, both concurrent operation are executed during the merge and the updated or moved element, which is referenced ("used") is annotated with a dedicated warning.

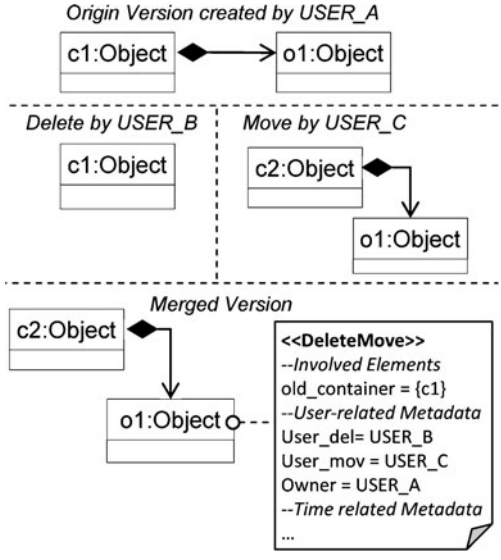

Figure 9. Merge Rule for Delete/Move Conflict.

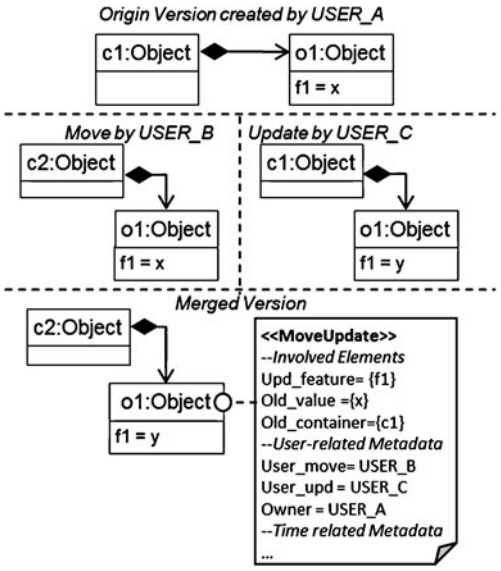

Figure 10. Merge Rule for Move/Update Warning.

7.1.1. *N-way merge rules*

In the previous subsection, we have presented different merge rules, which are applied if a conflict arises between two users. This so-called three-way merging is insufficient when three or more users perform their changes to the same model in parallel following the conflict-tolerant merge approach. For instance, when two users rename the same model element differently, the original value remains in the merged model and both update values are kept in the dedicated conflict annotation. If a third user also renames the same element differently, no conflict would be reported. Thus, in the following, we present merge rules, which are applied, when three or more modelers change the same model element in parallel. When merging in a conflict-tolerant way and two or more users participate, it is important that also the already reported conflicts in terms of annotations are checked. For instance, when a Delete/Update conflict occurs between two versions, the update is executed, the deletion is omitted, and the dedicated model element is marked as conflict. If a third user has also updated the same model element in parallel, the merge rules presented in Section 7.1 would not match. Therefore, the conflict detection must also check the applied annotations of the changed attribute.

Furthermore, we want to present the modelers the conflicts in an aggregated way to avoid an overload of similar conflict annotations applied on the same element. For instance, if a model element is already annotated by a Delete/Update conflict and a third modeler also changes the same element in terms of an update, a Delete/Update and an Update/Update conflict would additionally be reported. Thus, we *extend* the first conflict annotation resulting in only one Delete/Update

annotation containing information about one deletion and two updates. Each change type can be instantiated as often as necessary. This means, that in the case described before, any number of deletes or updates might be performed on one element, but only one conflict annotation is applied.

In the following, we present two example merge rules where (i) a Delete/Update conflict and (ii) an Update/Update conflict are extended by a further update of the same model element performed by a third user.

In Figure 11, a merge rule is depicted for the following scenario: User B has updated the feature f1 of object o1 from the value x to y, which was deleted by User C, leading to a Delete/Update conflict as already discussed in Section 7.1 and depicted in Figure 6. However, User D has also updated the feature f1 to the value z in parallel. Thus, with this rule the already applied Delete/Update conflict annotation is extended by including the update value and additional information of User D. This rule is needed, because it has to be checked, whether the updated object has already been deleted by analyzing all already applied annotations of this object. Since the object o1 has already been deleted as indicated by the first Delete/Update conflict, the update of User D has to be seen as a concurrent change. The Delete/Update conflict annotation is extended to present the users the conflicts in a minimal and more comprehensible way as depicted at the bottom of Figure 11.

As mentioned before, the Conflict-tolerant Merge ensures that the changes of *N* users are considered when merging. For example, if a fourth user would also

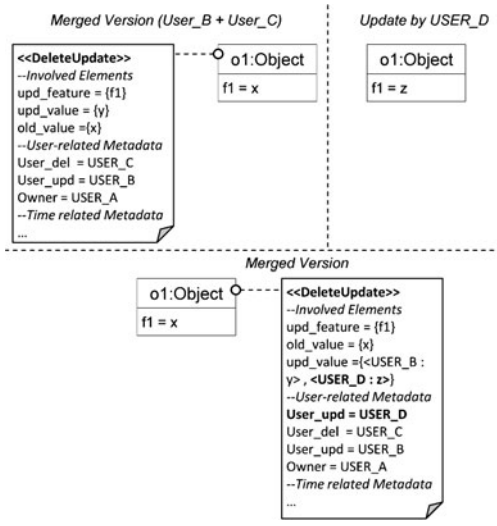

Figure 11. Merge Rule for a Delete/Update/Update Conflict.

concurrently update the same feature, the procedure is repeated and the applied annotation is extended again. In the following, we present another merge rule as depicted in Figure 12. In this scenario, three users perform three concurrent updates of the same model element.

Users B and C has concurrently updated the feature f1 of object o1. The merge rule for this Update/Update conflict is depicted in Figure 5. The original value x of f1 remains in the merged model and the update values y and z are included in the conflict annotation. Now, User D has also updated the f1 to the value a in parallel to User B and C. Although two more Update/Update conflicts are reported, the already applied Update/Update conflict annotation is extended by the third update. This annotation now contains all three updates.

Furthermore, it might happen that three or more different change types are performed on the same model element. Thus, beside the conflict pairs presented before, the following aggregated conflicts and warnings have to be introduced. The order of the individual change types of each combination does not matter and, for each change type, any number of changes can be included in the conflict annotation as discussed above.

- Delete - Update - Use Conflict
- Delete - Move - Use Conflict
- Delete - Move - Update Conflict
- Delete - Update - Use - Move Conflict
- Update - Move - Use Warning

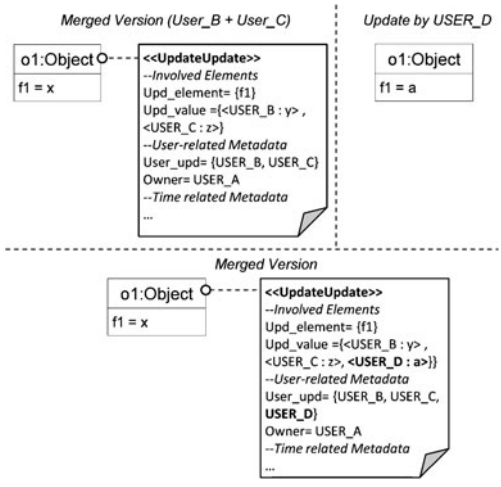

Figure 12. Merge Rule for a Update/Update/Update Conflict.

For instance, if a Delete/Update conflict annotation is already applied due to the changes of User A and B and User C now "uses" this Element by setting a reference, the conflict annotation is extended to indicate a Delete/Update/Use Conflict. If a User D would also perform a use operation in parallel, the Delete/Update/Use conflict annotation would be extended by including the second use operation as presented by the examples before. Similar to this scenario, all other combinations can be build to ensure that N parallel versions of a model can be merged in a conflict-tolerant way and that the conflicts are presented to the users in an aggregated view.

7.2. The merge algorithm at a glance

In this subsection, we present an overview on the conflict-tolerant merge algorithm. Figure 13 depicts a UML Class Diagram including all classes used in this algorithm. The diagram contains a class Model representing a versioned software model. A Model contains one root ModelElement, which again might contain several child elements. Each model element has an ID and knows its container model element. Model elements may be enriched with several Annotations and can be further described by MetaInfos which contains information about the users who recently changed the model elements, the Changes themselves, and the base version. The aggregated annotations presented in Section 10 are not depicted for reasons of clarity.

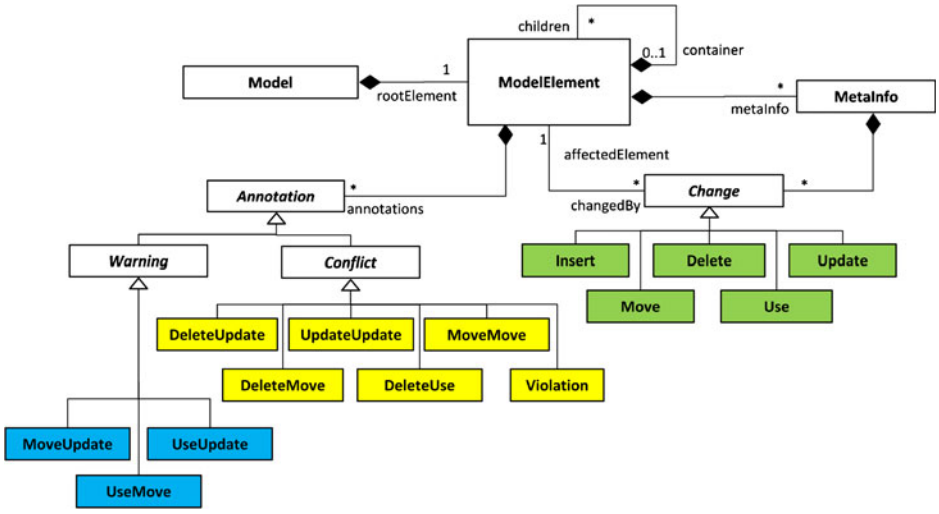

Figure 13. Changes and Annotations at a Glance.

The algorithm consists of two phases, in particular, the *fusion* phase and the *validation* phase. In the *fusion* phase, the algorithm iterates over all changes and checks for conflicts by using the afore presented conflict rules. Once the conflicts are identified, a copy of the common origin model of the modified models is created and all non-conflicting changes are applied to this copy. Next, conflicting changes are considered and merged as presented in the previous section. Thereby, all conflicts are resolved by prioritizing updates and moves over deletions and contradictory updates are omitted. Furthermore, annotations indicating the conflict and providing all relevant meta-data are immediately added to the involved model elements. This allows us to fully automatically produce a merged model in any case which is additionally enhanced with conflict information based on annotations.
In the *validation* phase, the merged model is validated. Validation means to reveal violations of rules and constraints defined by the modeling language. Thus, we reuse model validation frameworks which are able to detect language constraint violations in the merged model. Like in the fusion phase, we annotate all model elements which are subject to violations in this phase. Therefore, we iterate through all revealed violations and annotate the elements involved in the violation by so-called Violation annotations.

7.3. Merging the running example

In this subsection, the afore presented merge algorithm is applied to the running example (cf. Figure 3) in order to illustrate its function in more detail. In contrast to the standard merge depicted in Figure 3, where conflicts are resolved immediately comparing changed models pairwise, Figure 14 illustrates the merge results obtained using the *Conflict-tolerant Merge* and the annotations of the elements marked with the corresponding number.

7.3.1. *Merging V1a with V1b into V2*

In the example introduced in Section 5, Sally checks in after Harry has committed his changes to the common repository. His changes comprise the insertion of the attribute color in class Car, and three updates. He updated the name of attribute bday to birthday in class Employee, also the name of the attribute type to carType, and changed the aggregation type of the reference connecting Employee and Car from unspecified to composition. Consequently, the classes Car and Employee as well as the reference have to be considered as updated when Sally's version V1b is merged with the latest version V1a of Harry.

Fusion phase Sally's changes comprises a move of the attribute type, a deletion of class Car, and a deletion of the reference connecting Employee and Car.

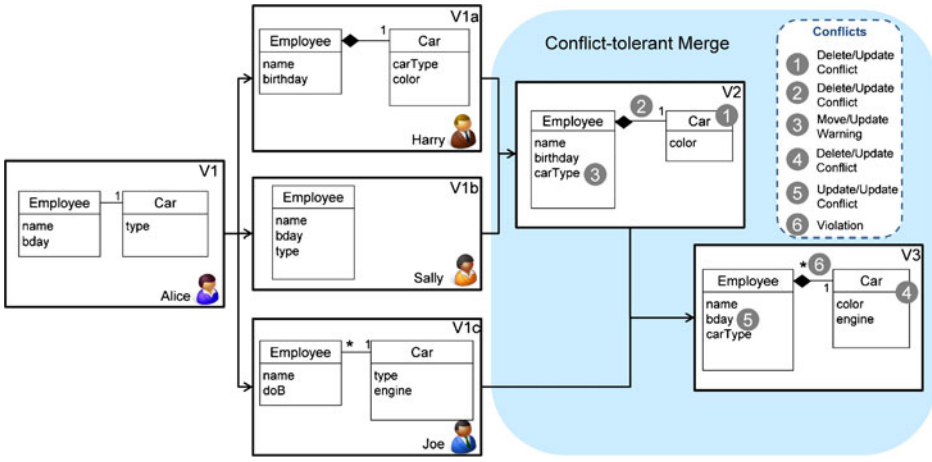

Figure 14. Conflict-tolerant Merging of the Running Example and Annotated Conflicts.

Subsequently, the merge algorithm iterates over all changes of Sally and checks for each changed element if an afore presented merge rule has to be applied. For the first change, i.e., the move of the attribute type, the conditions of the "Move/Update rule" match, because this element has been concurrently updated by Harry. Since the rule executes both operations and annotates the element with a Move/Update Warning, in the merged version V2 the attribute carType is contained by the class Employee and annotated with the according conflict. The second change of Sally is the deletion of the class Car leading to a Delete/Update conflict, because the same class has been updated by Harry by inserting the attribute color. Thus, the Delete/Update merge rule applies so that the deletion of the class Car is ignored, the insertion of the attribute is performed, and the class is annotated with a Delete/Update Conflict annotation. The same is true for the third change of Sally, i.e., the deletion of the reference between the classes Car and Employee: the deletion of the reference is ignored but is updated to a composition and a Delete/Update Conflict annotation is applied to the reference. Since Harry has updated the name of the attribute bday to birthday and Sally did not perform any operation on this element, no special merge rule matches and, thus, the update of Harry is performed without annotating the attribute.

Validation phase Now the merged version is validated but since there are no language violations in this model, the merge algorithm terminates and the merged version is published into the central repository (cf. V2 in Figure 14).

7.3.2. *Merging V2 with V1c into V3*

According to Figure 3, Joe checks in his version V1c, which has to be merged with the head version V2 in the repository.

Between V1 and V2, Harry and Sally performed several changes, which are encompassed in V2. Recall that Harry updated the attribute bday as well as the type of the reference from Employee to Car. Moreover, he added an attribute to Car. Sally removed the class Car and also the reference between Employee and Car.

Fusion phase The first change of Joe comprises the renaming of the attribute bday to doB. Since this attribute has also been updated concurrently by Harry, the condition of "Update/Update" merge rule matches. Consequently, the rename of Harry is reverted to its original name bday and an Update/Update Conflict annotations is applied. Recall that the new values of this element, i.e., birthday and doB, are not deleted. Instead, they are saved as meta information in the annotation. The next change of Joe concerns the insertion of the attribute engine into the class Car. Since the class Car is already attached with a Delete/Update Conflict annotation, this annotation is extended containing also the information about the update operation of Joe. The final change of Joe concerns the update of the reference multiplicity which, in particular, has been set to unbounded (notated as *). Since no merge rule matches, this operation can be performed without causing a conflict.

Validation phase After all changes have been handled, we may move on to the validation phase. When the merged version of the model V3 is validated, a language violation is reported. References of type Composition may not have an unbound multiplicity. Consequently, each element involved in this violation is marked with a Violation annotation. In our example, only the reference is involved and annotated.

8. Consolidation

After all developers have finished to contribute changes to the repository, the all-encompassing head revision in the repository might contain several conflicts and inconsistencies. At a certain point in time in the software development project, a consolidated model version reflecting a unified and consistent view on the modeled domain has to be found. For this step, we aim at addressing the requirements from practice as indicated by the interviewed experts (cf. Section 4). According to the interviewed experts, the versioning system should allow for *resolving conflicts collaboratively* and maintain all information that is necessary to *comprehend the decisions* taken for *finding the consolidated version*. In the

following, we present the necessary techniques to realize these requirements based on the merged model and the conflict annotations created by the conflict-tolerant merge discussed in the previous section.

8.1. Conflict resolution

For supporting the consolidation phase, we have developed a dedicated structural model defining the relevant information about a conflict resolution as well as a dedicated behavioral model for formalizing the life cycle of conflicts. The resulting *Conflict Resolution Model* is depicted in Figure 15. A ModelElement may be annotated by a conflict as already discussed in the previous section. A conflict may be assigned to different users, which are responsible to resolve the conflict. These users may propose different resolutions, but exactly one of these resolutions has to be finally accepted in order to resolve the conflict. Two possible kinds of resolution strategies exist: (1) either select *one* out of the conflicting changes, or (2) discard both and perform a custom resolution, which may contain several changes. In the latter case, the modeled resolution is stored as its own Diff to comprehend afterwards what happened to the conflict in the resolution process.

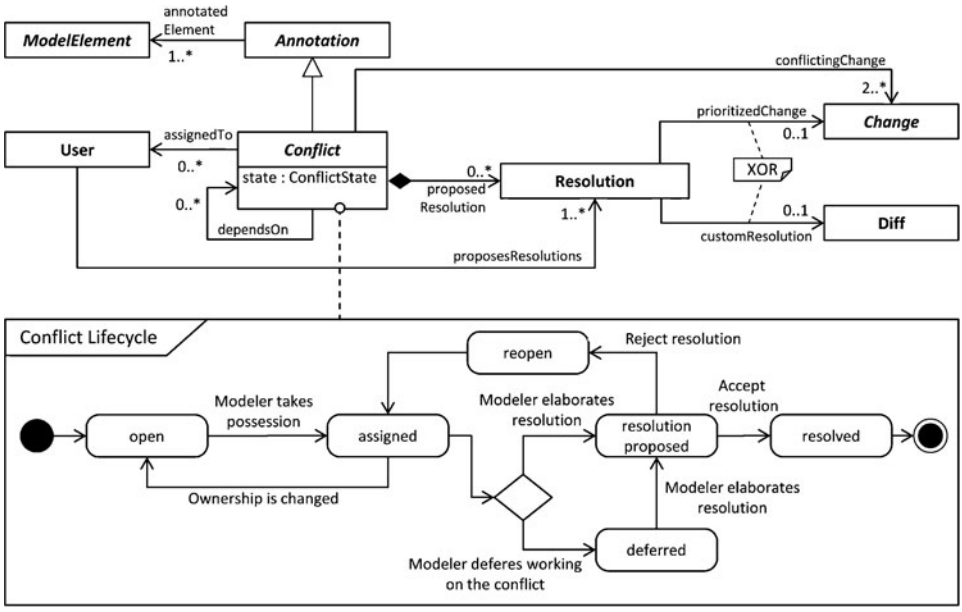

Figure 15. Conflict Resolution Model with Conflict Lifecycle.

Furthermore, a conflict may *depend* on another conflict: Coming back to the example depicted in Figure 14, a Delete/Update conflict is already annotated on class Car, because Sally has deleted this class which is updated by Harry by introducing the new attribute color. Since Joe has also introduced in parallel the attribute engine in the same class, a further Delete/Update is reported, but, as described before, the conflict annotation is integrated in the Delete/Update conflict annotation already applied on the class Car. The resolution of one conflict is now dependent on the resolution of the other conflict. In our example, if the class Car should remain in the model, both independent updates, i.e., insertion of attribute color and engine are prioritized if they are not contradicting. More investigations on how to automatically resolve dependent conflicts will be conducted in future work and, thus, we do not go into more detail in this paper.

In the conflict resolution process, a conflict passes through different states that is comparable with the life-cycle of a bug in a dedicated tracking system such as Bugzilla.[3] At the bottom of Figure 15 these states are depicted with the help of a UML State Diagram: When a conflict is reported it is in the state open. Now, modelers can be assigned to the dedicated conflict. These modelers are not necessarily the same as those who have caused the conflict, but may also include the modeler who has created the model element. After that, modelers may now elaborate the resolution or explicitly defer working on the conflict. Therefore, there is the additional state deferred that is important to make the other modelers aware of the deferral of the conflict resolution if, for example, more time is needed to resolve the conflict. As soon as a resolution has been elaborated by the assigned modelers, a conflict is in the state resolution proposed and can be reviewed by the others. If this resolution is rejected, the conflict can be reopened to start a new resolution phase. In the opposite case, the conflict is in the state resolved which directly leads to the end state of its life-cycle.

In Figure 16, the resolution process is depicted for a concrete Delete/Update conflict. In this example, the deletion of the class Car is conflicting with the insertion of the attribute color leading to a Delete/Update conflict. Since our conflict-tolerant merge does not execute conflicting delete operations, but only marks the involved elements as deleted, the class Car remains in the merged version of the model and the attribute color is also included. In the first step, the class is annotated with a dedicated annotation and the state of the conflict is open. In the second step, the user Harry is assigned to the conflict, who is now responsible to resolve it. He proposes a new resolution, by selecting the update of the class Car. This is represented in the model by having a link from the resolution object to the prioritized change, i.e., the update object in our example.

[3] http://www.bugzilla.org/docs/2.18/html/lifecycle.html

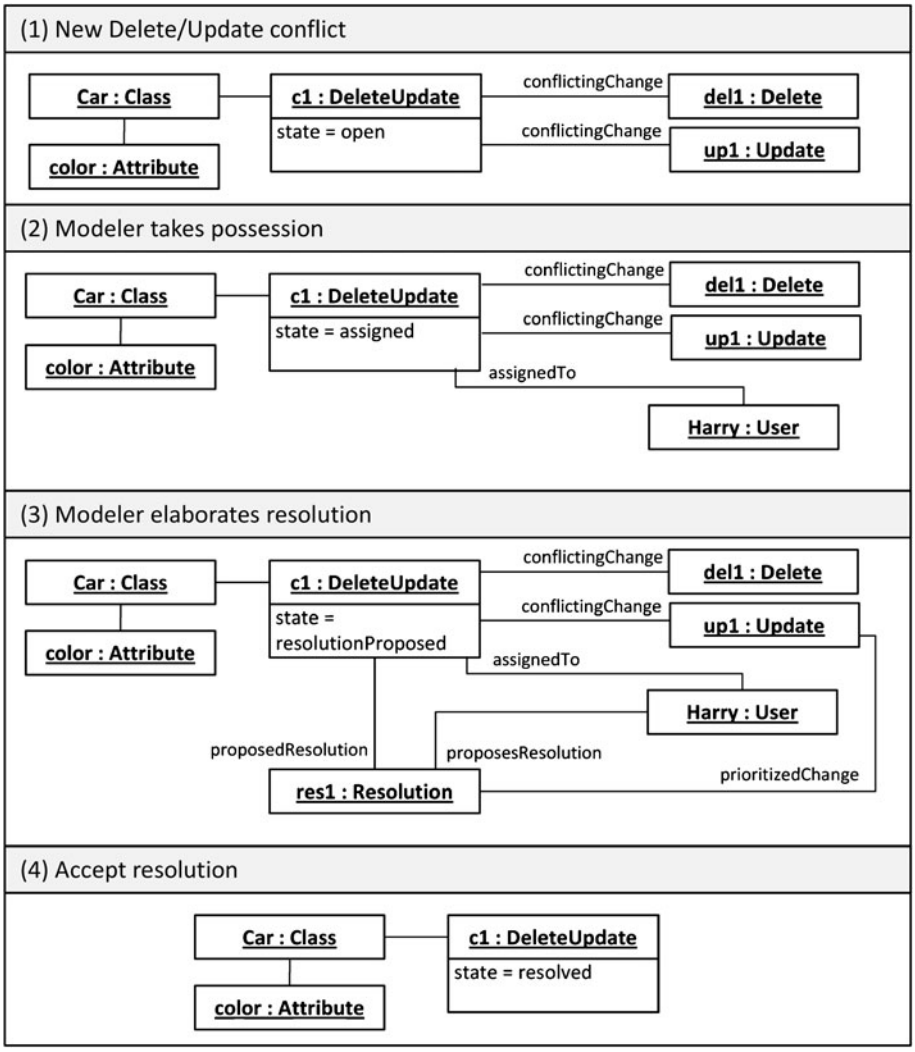

Figure 16. Delete/Update Conflict Resolution Example.

After this resolution proposal is reviewed by others and finally accepted, the merged version still contains the class Car with its attribute color. Finally, the conflict annotation is hidden but still available in the central model repository for provenance reasons.

In the following, a second example depicted in Figure 17 is presented where the resolution of an Update/Update conflict is conducted. Harry and Joe have concurrently changed the name of the attribute bday, which was created by Alice, to birthday and doB, respectively. Thus, in the merged version the attribute has its origin name bday and it annotated with an Update/Update conflict. The state of

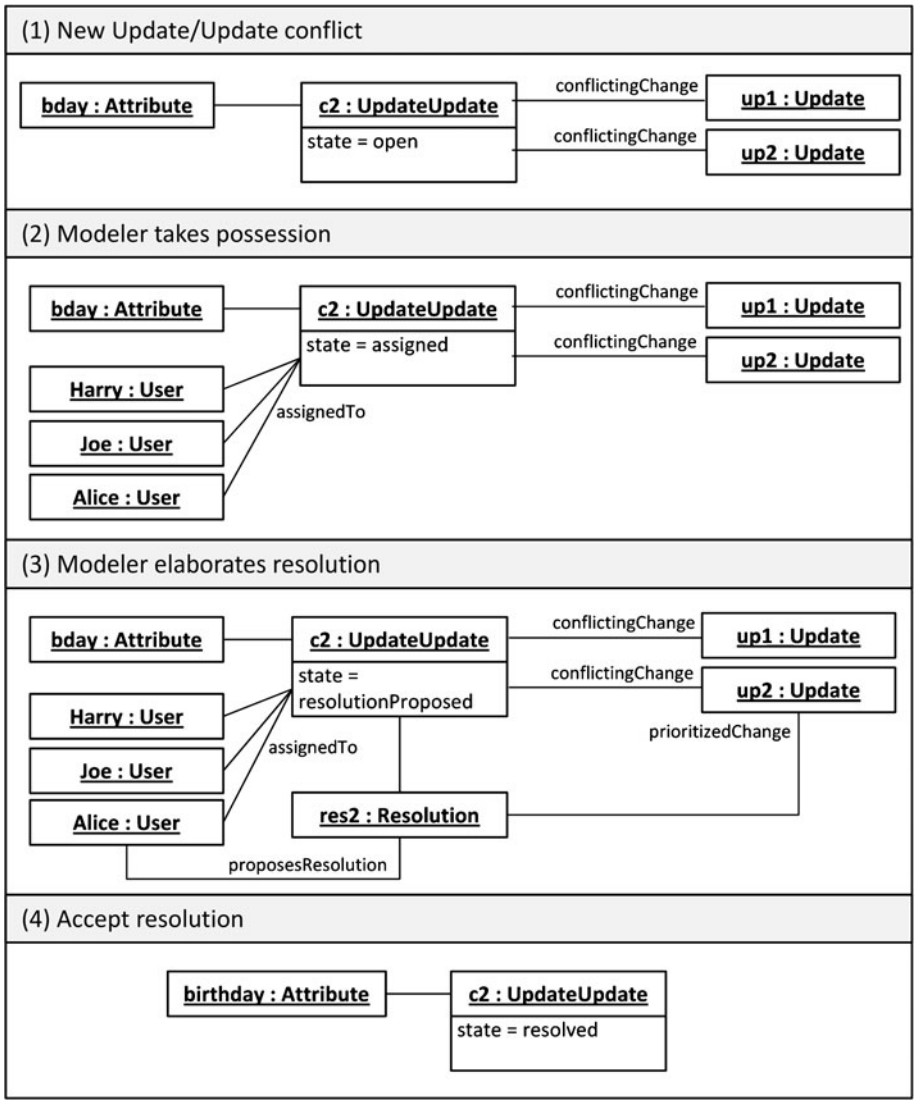

Figure 17. Update/Update Conflict Resolution Example.

the conflict is open. In a second step, not only Harry and Joe are assigned to this conflict to resolve it, but in this concrete example also Alice, because she is the creator of the model element. Thirdly, since birthday is a more proper name for an Employee's birthday, they decide to prioritize the update of Harry, i.e., Up2. Note that more than one modelers may propose more than one resolution proposal. However, in this example only Alice explicitly proposes the resolution of the conflict. Finally, the resolution proposal is accepted and the update is performed. Now, the attribute is named birthday.

The state of the conflict is changed to resolved and, as mentioned before, it is hidden but kept in the repository.

8.2. Consolidating the running example

The *consolidation phase* is supported by an adequate visualization of all conflicts in the unconsolidated model. This view serves as a basis to discuss existing issues and different points of view.

Coming back to our running example, Harry, Sally, Joe, and Alice collaborate to resolve all existing conflicts. Different possibilities exist how the developers are able resolve the occurred conflicts collaboratively. In this paper, we presented the possibility of asynchronous conflict resolution, where the assigned modelers can prioritize one of the conflicting operations. In Brosch et al. (2009) we proposed synchronous conflict resolution, where the modelers may remotely discuss and resolve the conflicts together. Also for this kind of resolution, the conflict annotations build a good basis for discussion. When evaluating the Conflict-tolerant Merge as presented in Section 10, we used face-to-face session for resolving the occurred conflicts. In the following, we discuss a potential collaborative resolution for each conflict in the running example in more detail. Supported by the conflict report and the provided metadata, they find a consolidated version which is depicted in Figure 18 by resolving the following six conflicts.

1. *Delete/Update: Class Car.* First of all, the modelers have to decide whether the class Car is needed. Since Sally was of the opinion that exactly one car is assigned to one employee, she has deleted the class Car and has inlined its attributes to the class Employee to ensure the efficient querying of information. However, after Harry and Joe communicate their opinions, all participants are convinced to keep the class Car in the model. Due to this decision, both attributes, i.e., color and engine, remains in the class Car.

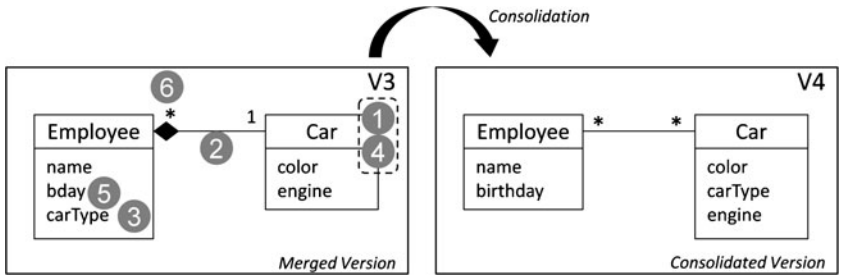

Figure 18. Consolidated Version by Turning Conflicts into Collaborations.

2. *Delete/Update: Reference employee-to-car.* Since class Car is not deleted, the reference between both classes is also retained.
3. *Move/Update: Attribute carType.* The assigned modelers also check the occurred warning and notice that the attribute carType is now in the "wrong" class. Therefore, it is moved back to its origin class Car.
4. *Update/Update: Attribute bday.* Harry and Joe have concurrently renamed the attribute bday, which has been introduced by Alice. Both agree, that bday and doB may lead to misunderstandings and, therefore, decide to use birthday.
5. *Violation: Reference cardinality.* Harry has introduced the containment relationship between the two classes and Joe has set the multiplicity to "unbound" leading to a metamodel violation. They discuss the different possibilities and decide to resolve it manually as depicted in Figure 19. In the final version, a company car can be assigned to more than one employees and one employee can be assigned to more than one car. Thus, the containment relationship is converted back to a non-containment relationship and both multiplicities are set to "unbound".

After Harry, Sally, Joe, and Alice have finished the resolution of all conflicts and, furthermore, all resolution proposals are accepted by the responsible modelers, the consolidated version of the model is then saved in the repository as new version *V4*.

The presented example illustrates that it is highly beneficial to conjointly discuss each conflict because there are different ways to resolve them due to the different viewpoints of the involved modelers. Conflict resolution is an error-prone task when only one modeler has the full responsibility to resolve conflicts. Our presented approach counteracts this problem. We retain all information necessary to make reasonable resolution decisions and, in addition, collaboration and discussions are fostered. The resulting final version (cf. Figure 18) reflects all intentions much better than this could have been achieved by one modeler on her own. In summary, such a consolidation leads to a unification of the different viewpoints and finally to a model of higher quality accepted by all team members.

9. Implementation

To evaluate the afore presented concepts, we created a prototypical implementation which is outlined in this section. Our implementation is realized as Eclipse[4] plug-in and is build upon the Eclipse Modeling Framework (EMF) (Budinsky et al. 2003), one of the most adopted (meta-)modeling frameworks in practice. Due to this widespread adoption, a pantheon of modeling languages are specified in

[4] http://www.eclipse.org

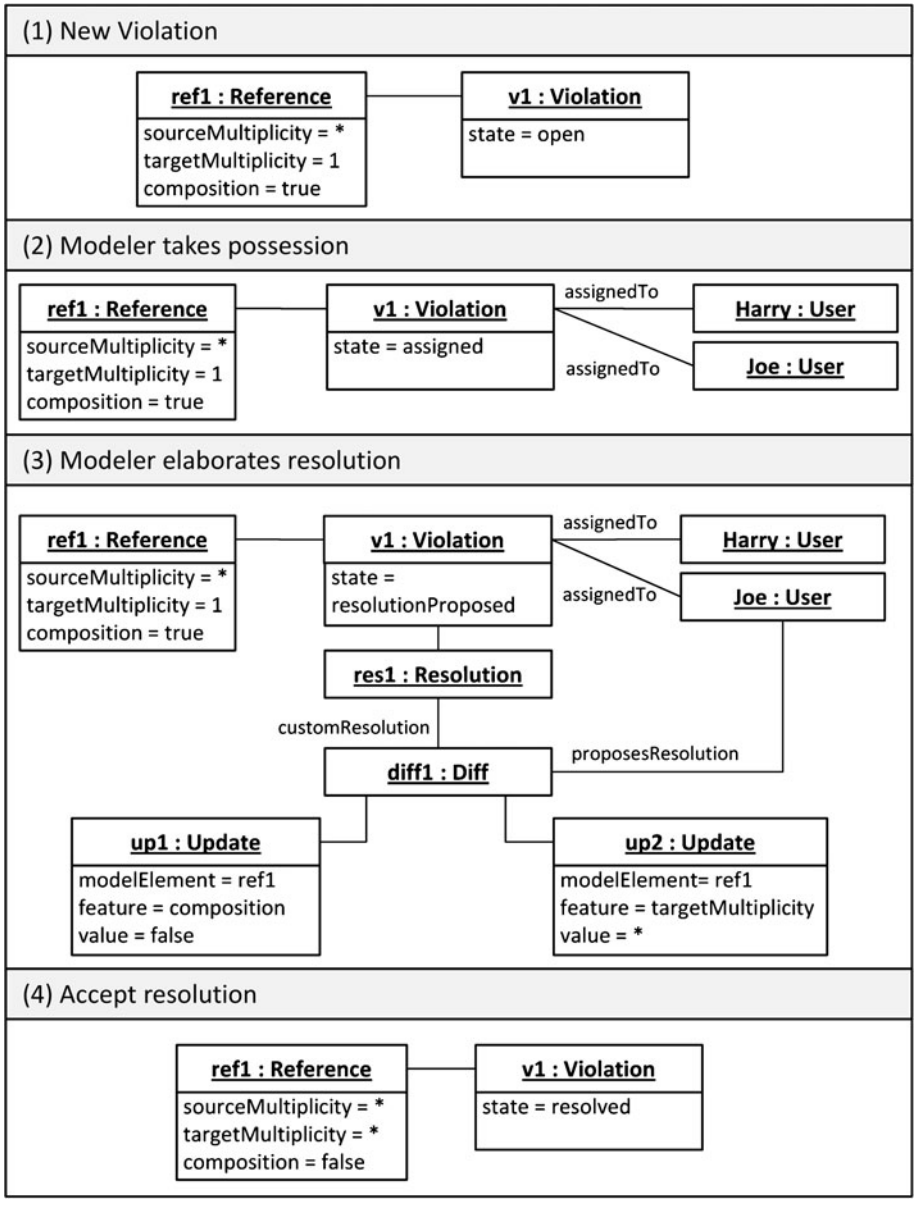

Figure 19. Violation Resolution Example.

EMF—among them UML 2.0. Using the powerful reflection mechanisms of EMF, we designed our implementation to support every EMF-based modeling language. In the following we shortly outline the implementation of all components required to realize the presented concepts.

Detecting changes To identify the changes applied between an origin model and a revision of it, two different approaches exist. On the one hand, *state-based* approaches take two versions of a model as input and compute the model differences by comparing these two model states (Mens 2002). On the other hand, as introduced by Lippe and van Oosterom (1992), *operation-based* approaches obtain the changes by directly recording them in the modeling environment as they are performed by the user. Both approaches have their advantages and disadvantages. In comparison to state-based 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 state-based approaches. 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 must represent them in a common format. To avoid depending on the used editor, we conduct state-based model differencing based on the extensible model comparison framework EMF Compare (Brun and Pierantonio 2008). To increase the precision of EMF Compare, we implemented an extension to EMF Compare exploiting *universally unique identifiers* (UUIDs) which are attached to each model element and which are, once assigned, never changed anymore. Thus, using UUIDs also moved and intensively changed model elements can be recognized across two versions of the same model which allows for a more precise computation of applied changes. Once, the changes between an original model and two revisions of it are identified, they are saved to two so-called *difference models* and passed to the conflict detection component.

Detecting contradictory changes Having identified the precise changes concurrently applied by two users, we may now proceed with detecting conflicts between them. In Section 7, we introduced rules for detecting, annotating, and merging conflicts. These rules involve a change pattern which specifies two changes concurrently applied to the same origin model element (cf. upper left area of Figures 5, 6, 7, 8, 9 and 10). Hence, as soon as such a pattern exists in the difference models, a corresponding conflict is at hand. For instance, having two *updates* changing the same model element, a conflict shall be raised (cf. Figure 5). Having these rules, implementing the conflict detection component is largely straightforward. Generally speaking, for all change combinations of both difference models it has to be checked whether one of the aforementioned conflict patterns matches and raises, in case a match is at hand, a proper conflict. However, for the sake of efficiency, we refrain from checking the complete cross-product of all change combinations between all changes of both difference models. In contrast, both difference models are translated in a first step into an optimized view which sorts all changes based on their type into potentially conflicting combinations. Secondly, these combinations are triaged whether they spatially affected overlapping parts of the original model. Finally, all remaining combinations are checked in detail by evaluating the rules presented before.

Once, all conflicting combinations of changes are identified, they are marked accordingly to be regarded in the following step.

Merging models After all conflicting changes have been identified, we may create a merged model according to the merge rules specified in Section 7. In a first step, a copy of the common origin model is created. Of course, in this copy the UUIDs of the origin model must be retained. Next, all non-conflicting changes contained by both difference models are reapplied to the created copy. Therefore, we implemented a dedicated model transformation engine based on the *merging framework* of EMF Compare (Brun and Pierantonio 2008) which is able to transform a model according to the afore detected model differences. The model elements to be transformed are identified using the aforementioned UUIDs. In the next step, we handle conflicting changes according to the merge rules presented in Figures 5, 6, 7, 8, 9 and 10, i.e., omitting contradictory updates and moves as well as prioritizing updates over deletions.

Detecting violations Besides conflicting changes, the merged model may incorporate violations of the conformance rules of the respective modeling language. For detecting such violations, we use the EMF Validation Framework.[5] Thereby, 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 expressed in OCL or Java. Whenever a violation is detected, diagnostics are returned which describe the severity of the constraint violation and provide an error message as well as the model elements involved in the respective violation. This information is used to mark violations besides overlapping changes in the merged model in the next step.

Annotating models Finally, the merged model is enriched with the conflict and violation annotations as specified by the merge rules in Figures 5, 6, 7, 8, 9 and 10 conforming to the annotation metamodel depicted in Figure 13. Unfortunately, EMF does not provide an inherent annotation mechanism for enriching a model with additional information. Therefore, we ported the lightweight extension mechanism known from UML Profiles (Fuentes-Fernández and Vallecillo-Moreno 2004) to the realm of EMF models as presented by Langer et al. (2012). Using this tool called EMF Profiles, every EMF-based model may be annotated with stereotypes containing tagged values. If for instance, an *Update/Update* conflict appeared (cf. Figure 5), the corresponding stereotype UpdateUpdate is applied to the object which was concurrently modified. This stereotype contains information—in terms of tagged values—on the contradictory updated values as well as the users who performed the conflicting changes. Stereotype applications may be visualized on top of the graphical representation of a model to support the user in resolving all annotated conflicts directly in the model. Annotations are saved in a separate model

[5] http://www.eclipse.org/modeling/emf/?project=validation#validation

to avoid polluting the merged model. However, when checking-in the merged model comprising conflicting changes, the annotation model is saved alongside the merged model to also allow other modelers to investigate and resolve existing conflicts.

Consolidation As discussed in Section 8, users may resolve conflicts by applying either one of the two changes or by performing a new custom set of changes. All of these options may be performed directly in the modeling editor. We exploit the SelectionListener interface which is implemented by all EMF-based editors to allow for displaying all annotated conflicts in a dedicated view whenever a model element is selected by the user in the modeling editor. In this dedicated view, the user may now choose how to resolve the displayed conflict. Once this is done, the tagged value representing the state of the conflict is changed accordingly. After all or a subset of all conflicts are resolved, the model alongside the updated annotation model is again checked into the repository. As argued earlier, it is often desired by developers to collaboratively resolve certain conflicts in a synchronous manner in contrast to resolve a conflict by oneself (Brosch et al. 2009). Although this has not been integrated in our implementation yet, users may use *Connected Data Objects*[6] (CDO), a distributed shared model framework which allows to simultaneously edit EMF-based models over a network. Since the actual model as well as the annotation model is based on EMF, also a distributed collaborative resolution of existing conflicts is possible by using the CDO technology.

User interface Coming back to the running example introduced in Section 5 and depicted in Figure 14. A screenshot of Eclipse consisting of the merged model is depicted in Figure 20, in which the parallel versions of Harry, Sally, and Joe are merged into one version. The occurred conflicts, are automatically annotated using EMF Profiles and are visualized with dedicated icons. If a user selects an annotated model element, the applied conflict annotation are presented in the "Profiles Applications" view. In the screenshot, the Update/Update conflict applied on the attribute bday of the class employee is selected. At the bottom of this screenshot, the tagged values of this stereotype are presented in terms of properties. The users may now see that Harry and Joe have concurrently renamed the attribute to birthday and doB, respectively. Both users are also assigned to this conflict and have chosen to prioritize the update of Harry. Thus, the status of the conflict is set on resolution proposed.

10. Evaluation

In this section, we report on our initial evaluation of the Conflict-tolerant Merge presented in this paper in comparison with EMF Compare,[7] the state-of-the-art tool for three-way model comparison and model merging for EMF-based models. We

[6] http://wiki.eclipse.org/CDO
[7] http://www.eclipse.org/emf/compare/

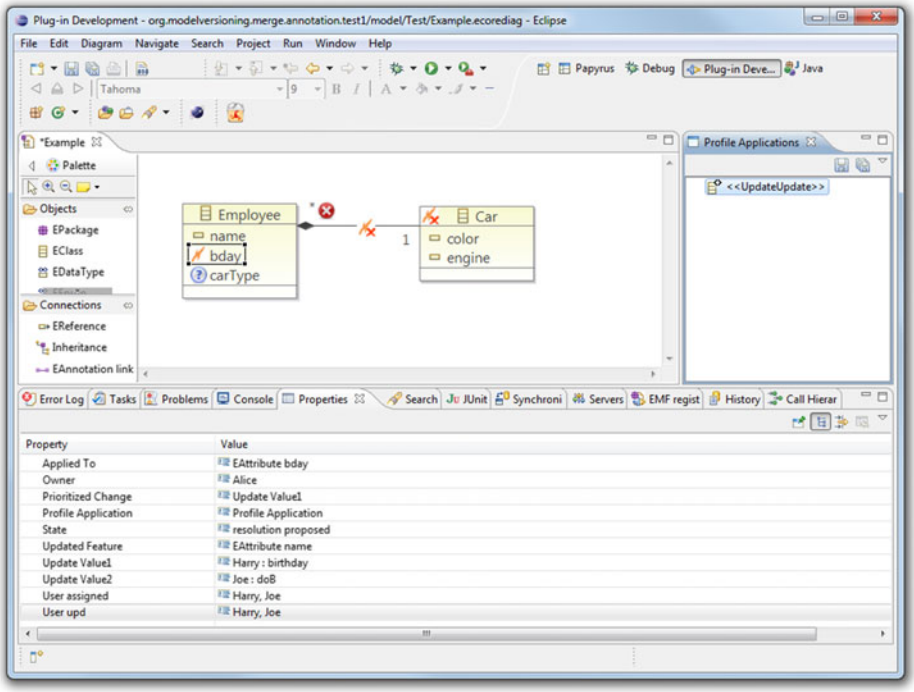

Figure 20. Conflict-tolerant Merge as Eclipse Plug-in.

decided to use EMF Compare, because it is part of the Eclipse Modeling Project[8] and freely available. We have done a quasi-experimental study (Campbell et al. 1963) with 18 participants and interviewed them afterwards with the help of a questionnaire. In the following, we describe the study in detail, discuss the challenges encountered by the participants, and present the results and our findings of this evaluation. The evaluation is based on observations and valuable feedback from our participants.

10.1. General setting

As already discussed in previous sections, the Conflict-tolerant Merge follows a new paradigm for developing models in teams. It offers an alternative to the traditional versioning paradigm, in which the developer who commits the changes is solely responsible for resolving the detected conflicts immediately. To simulate this iterative process, we used EMF Compare (version 1.1.2), which offers a tree-based representation of two versions of the model and correspondences between the detected conflicts. In contrast, the Conflict-tolerant Merge incorporates all changes of all developers, marks occurred conflicts with the help of EMF Profiles (cf. Section 9) and at a later point of time the conflicts are

[8] http://eclipse.org/modeling/

resolved leading to a consolidated model. In addition, this annotation mechanism offers the possibility to visualize the conflicts in the concrete, e.g., graphical syntax of the models.

With the help of this evaluation, we want to gain experiences of using both systems based on the following assumptions.

1. The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.
2. The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.
3. Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.

The evaluation of the approach was conducted with groups of participants, which had to change a pre-defined model in parallel and independently of each other with the help of concrete change requests. One individual group, i.e., one run, passed through different phases, which are described in the following subsection in detail. The whole procedure is repeated for each group.

10.2. Study procedure

In this study, we distinguished between two different roles: the observer and the participant. The authors of this paper assumed the role of the observer, who was responsible for preparing this evaluation and for the trouble-free execution of the study.

In total, 18 participants took part in the study. All of them had knowledge in modeling, did not already work with one of the tools, and have not been involved in the research project in which the presented approach has been developed to ensure that their answers are objective. We decided to invite participants with an industrial as well as research background. Three different "kinds" of groups were assembled by us: (i) purely scientific group, (ii) purely industrial group, and (iii) mixed group. In total, ten participants were working in industry and 8 in research institutions (Academia). The allocation of the participants to the individual groups is depicted in Table 2.

Table 2. Allocation of participants.

Group	Academia	Industry
1	0	3
2	3	0
3	3	0
4	1	2
5	0	3
6	1	2

Each group contained three participants (Participant A, Participant B, and Participant C) who had to perform predefined changes in parallel and independently. They were randomly allocated to these roles. They worked on a prepared model and had to merge these three versions including the resolution of occurred conflicts.

The whole procedure of the evaluation is depicted in Figure 21. One run with one group contained three phases and the duration of it was not restricted.

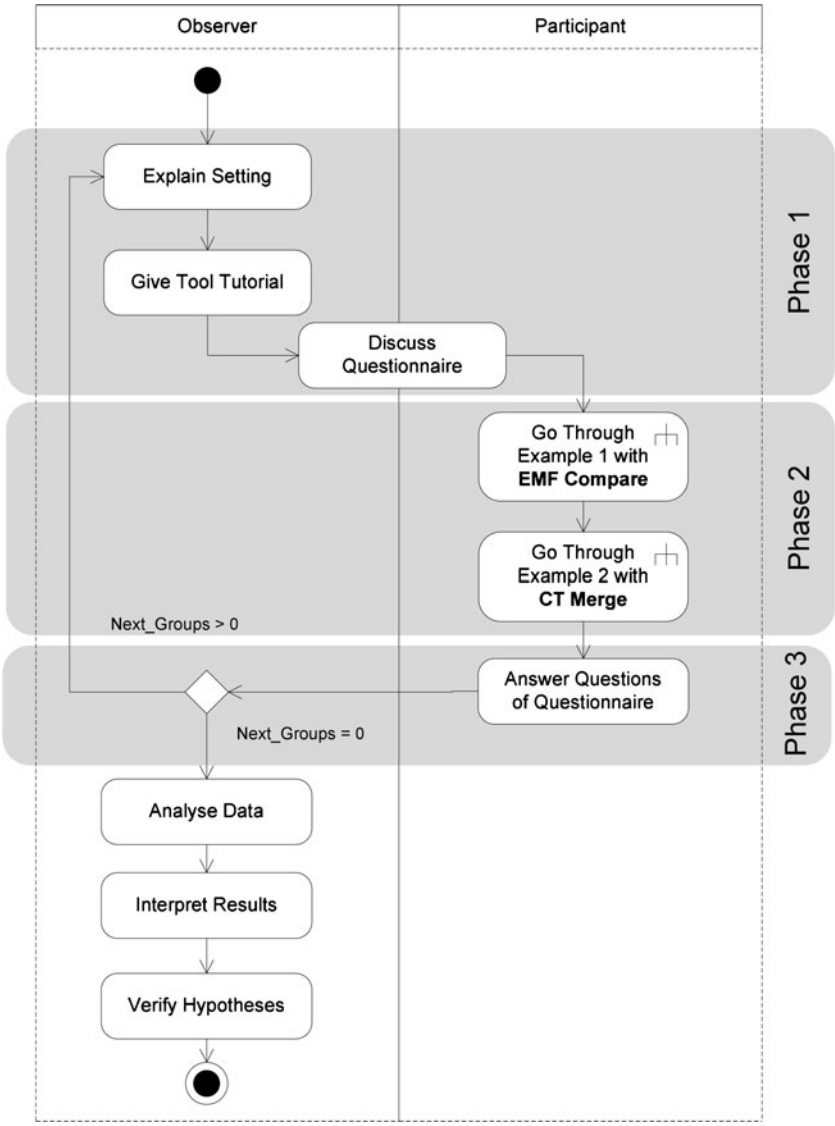

Figure 21. Study Procedure.

Phase 1: preparation The first phase started with an introduction, in which the general setting and the context of the evaluation were explained. The two different kinds of versioning paradigms and the two example models were presented. In the next step, the tools were presented to avoid that learning how to use a tool influences the evaluation results. After the short tool tutorials, the last step of phase 1 was to discuss each statement of the questionnaire, which was to be filled-in in phase 3 by the participants, to ensure that each question was not misinterpreted.

Phase 2: modeling For this phase, we prepared two different—but very similar—examples (Example A is depicted in Figure 23 and Example B in Figure 24). First, the participants have to go through one of the examples with EMF Compare and, secondly, through the other example with the Conflict-tolerant Merge. These two subprocesses of this phase are presented in more detail in the following.

The three participants got predefined change requests, which stated how they had to change the prepared models leading to three different versions of the model. The change requests for one participant contained almost four small modifications. The overall goal for the participants was the development of a merged version of each example, which was accepted by all participants.

The *first subprocess* describing the traditional versioning paradigm, which is simulated with the help of EMF Compare, is depicted in Figure 22(a). Participants A, B, and C changed the model in parallel and independently of

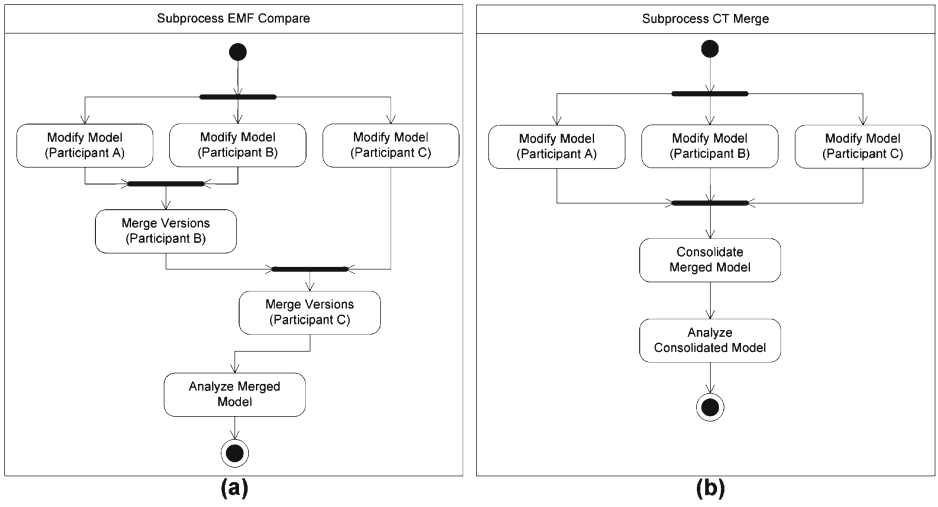

Figure 22. (**a**) Subprocess using EMF Compare and (**b**) Subprocess using CT Merge.

each other. First, Participant A had to check in her version resulting in a new head version in the central repository. When Participant B also tried the same, EMF Compare reported conflicts, which had to be resolved by her alone, when merging her version with the head version. When she finished merging the different versions of the model, a new head version was in the repository. Now, Participant C checked-in her changes and she had also to resolve conflicts when merging her version with the head version. At the end, all participants were invited to analyze the final result and to discuss the merged model.

After this process, the participants got another model and had to change it according to new change requests, which, again, are very similar to the change requests of the first example. Now, they used the Conflict-tolerant Merge tool to get a merged model. This *second subprocess* is depicted in Figure 22(b). Now, the participants did not have to resolve the conflicts immediately after they checked-in. The Conflict-tolerant Merge incorporated all changes of all three participants and annotated the occurred conflicts. After all participants finished changing the model, they discussed the merged version and resolved the conflicts together. After that, they analyzed the consolidated version of the model. For this study, we decided that the participants had to resolve the conflicts in a face-to-face session. However, the Conflict-tolerant Merge builds a good basis also for other synchronous and asynchronous conflict resolution approaches as discussed in Section 8 of this paper.

Phase 3: post-processing After the participants went through both examples, the post-processing phase was conducted. The participants had to answer the before discussed statements of the questionnaire and gave feedback during a final discussion. The questions and the discussion aimed to gather pros and cons of both systems and to get answers to the assumptions presented in Section 10.1.

10.3. Selection of examples

In the following, we shortly present the examples used in the study. We decided to take two different examples for each tool to ensure objectiveness. Furthermore, two different domains were chosen, i.e., cinema platform and e-learning system, with which all participants were familiar with. We also tried to keep the models quite simple and defined change requests, which were—in our opinion—not too complex.

10.3.1. *Example A*

Example A represents the original model of an e-learning system as depicted in Figure 23. This system consists of a class Teacher and Student; both inherit from

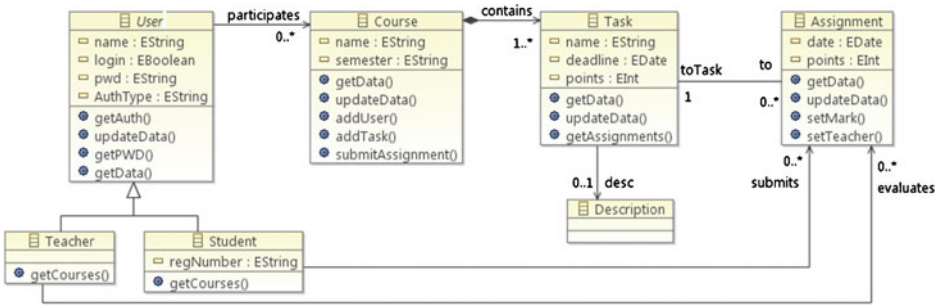

Figure 23. Example A: E-Learning System.

a common superclass User. A User may participate in a Course, which contains different tasks. Students may submit Assignments to a task, which are evaluated by a teacher.

The change requests for the participants and the resulting conflicts are presented in Table 3. "D/U" indicates a Delete/Update conflict, "U/U" indicates an Update/Update conflict, and a Move/Move conflict is represented by "M/M". In addition, a × indicates that a conflict between the two change requests is already marked with the dedicated abbreviation in the table.

Occurred conflicts

- Participant A changed the datatype of AuthType from String to Boolean, whereas Participant B created an enumeration containing different types of authorization. These concurrent changes led to an *Update/Update conflict*.
- Participant A performed the refactoring "Extract Class" on the attribute deadline of the class Task. She deleted this attribute, created a new class Duration, included the attributes start and end, and set a reference between the classes Task and Duration. In contrast, Participant B renamed the attribute deadline to end and included a new attribute start. Since the attribute deadline was deleted and updated in parallel, a *Delete/Update conflict* occurred.
- The final task of Participant A was to include the attributes abstract and details to the class Description. In contrast, Participant C deletes the class Description and, instead, she added an attribute description to the class Task. These concurrent changes led to *Delete/Update conflict*.
- Participant B moved the attribute login to the class Course whereas Participant C moved the same attribute to the newly added class Configuration. Here, a *Move/Move* was reported by the Conflict-tolerant Merge. EMFCompare could not report such a conflict. Instead, it duplicated the moved model element.
- Finally, Participant B and C concurrently renamed the attribute points in the classes Task and Assignment leading to *Update/Update conflicts*.

Table 3. Change Requests for Example A.

CR#	Change Request	A1	A2	A3	B1	B2	B3	B4	C1	C2	C3
Participant A											
A1	The system should only distinguish whether a user of the system has the necessary authorization or not.				*U/U*						
A2	Instead of a simple attribute deadline, the duration of a task with a start and an end date should be in an own class.						*D/U*				
A3	A description of a task should contain an abstract and additional details.								*D/U*		
Participant B											
B1	Different authorization types should be included like “admin”, “standard”, etc.	×									
B2	The login status of a user should be handled in the dedicated course.									*M/M*	
B3	A task should have a start and end date.		×								
B4	The difference between points of tasks and points of assignments should be more clear.										*U/U*
Participant C											
C1	The description of a task should not be an own class.			×							
C2	The login status of a user should be handled in an own configuration class.					×					
C3	The difference between points of tasks and points of assignments should be clearer.						×				

U/U ... Update/Update Conflict
D/U ... Delete/Update Conflict
M/M ... Move/Move Conflict
× ... Conflict already marked in the table

10.3.2. *Example B*

Example B represents a cinema platform as depicted in Figure 24. On such a platform, two different kinds of users exist, namely Visitor and Admin. Both inherit from the common superclass User. The administrator may create on the platform a new Cinema with its Rooms or a new Film, which can be rated by the visitors.

The change requests for the participants and resulting conflicts are presented in Table 4.

Occurred conflicts

- In this example, Participant A deleted the class Rating and added an attribute rating to class Film. In contrast, Participant B added to the deleted class an attribute stars and, thus, a *Delete/Update conflict* is reported.
- Participant A moved the attribute language of the class Room into the class Film, whereas Participant C moved the same attribute into another class, namely Cinema, leading to a *Move/Move conflict*.

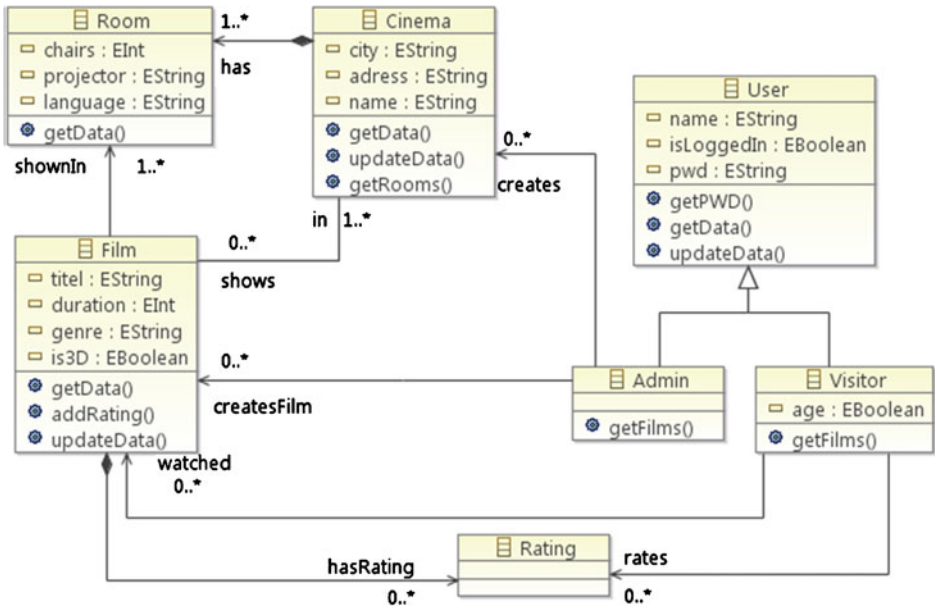

Figure 24. Example B: Cinema Platform.

Table 4. Change requests for example B.

CR#	Change Request	A1	A2	A3	B1	B2	B3	B4	C1	C2	C3
Participant A											
A1	The film rating should be an attribute of a film instead of an own class.				*D/U*						
A2	The attribute language should describe a film (not a room).								*M/M*		
A3	Rename the class room to hall.						*U/U*				
Participant B											
B1	A rating should additionally have an attribute, which represents the actual rating in terms of stars.	×									
B2	The seats of a room should be reflected in an own class to include the attributes row and number.									*D/U*	
B3	Rename the class room to "CinemaHall".			×							
B4	The age of a visitor should be represented as Integer value.										*U/U*
Participant C											
C1	The attribute language should describe a cinema (not a room).		×								
C2	Rename the chairs of a room to seats.					×					
C3	Three different types of the age of a visitor exist, namely "under12", "under14" and "under16".							×			

U ... Update/Update Conflict
D ... Delete/Update Conflict
M ... Move/Move Conflict
× ... *Conflict already marked in the table*

- Furthermore, Participant A and B renamed the class Room to Hall or to CinemaHall, respectively. This concurrent change led to a *Update/Update conflict*.
- Participant B deleted the attribute chairs and added instead a new class Chairs with the attributes row and number. In contrast, Participant C renamed the attribute chairs to seats leading to an *Delete/Update conflict*.
- Finally, Participant B changed the datatype of the attribute age of the class Visitor from String to Integer. In parallel, Participant C set the datatype of the same attribute to a newly added enumeration containing literals such as under12, under14, etc. These changes led to an *Update/Update conflict*.

10.4. Elaboration of questionnaire

After the examples were conducted with EMF Compare and with the Conflict-tolerant Merge, the participants had to rate different statements. Five answers each question were possible, namely "strongly agree", "agree", "disagree", and "strongly disagree". In addition, the answer "don't know" was also possible. The selection of the statements was based on two criteria. They should give us the possibility to (i) test the above mentioned assumptions and (ii) to check if objective and generally applicable criteria for the comparison of different data integration methods are fulfilled. These criteria are described in Batini et al. (1986) and are also suitable when merging different versions of a model in the context of model versioning, because the general problems are quite similar. Although the criteria can be taken as granted, all integration methods have to pit against them. In the following, we shortly describe them in the context of this evaluation:

- *Completeness:* If two versions of a model are merged, the resulting model should be complete, i.e., all changes of the participants should be incorporated in this model.
- *Correctness:* It is not enough that all changes of the participants are incorporated in the merged model. They also have to be incorporated correctly.
- *Minimality:* Although it has to be ensured that all changes of all participants are incorporated in the merged model, the model should also be minimal. This means that redundant elements should not be available in the merged model.
- *Understandability:* In addition, the changes of the participants, the occurred conflicts, and the merged model should be understandable. That means it should be comprehensible or obvious, which model element has been concurrently changed when regarding the resulting merged model.

In Table 5, the statements, which had to be rated by the participants are listed according to the assumptions.

Table 5. Statements to be rated by the participants.

S#	Statements of Questionnaire	C
Assumption 1: The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.		
1.	In contrast to EMF Compare, the CT Merge incorporates all of my changes.	Compl.
2.	In contrast to EMF Compare, the CT Merge incorporates the changes of the other developers.	
Assumption 2: The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.		
3.	The merged model of the CT Merge is more comprehensible than the one of EMF Compare.	Understandability
4.	The changes and resulting conflicts are more comprehensible when using the CT Merge than merging with EMF Compare.	
5.	The changes and resulting conflicts are more comprehensible if they are visualized in a merged model in terms of annotations than being visualized between different versions of the original model in terms of correspondences.	
Assumption 3: Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.		
6.	All of my changes are incorporated more adequately in the final model of the CT Merge than in the one of EMF Compare.	Correctness / Minimality
7.	I do more agree with the changes of the other developers incorporated in the final model of the CT Merge than in the one of EMF Compare.	
8.	Changes incorporated in the final model of the CT Merge are less redundant than in the one of EMF Compare.	
9.	All changes incorporated in the final model of the CT Merge are more correct than in the one of EMF Compare.	

S# ... Statement Number
C ... Data Integration Criteria
Compl. ... Completeness

10.5. Results

In the following, we present the results of the questionnaire and the most important observation as well as a critical discussion about this study.

Table 6. Results of questionnaire.

Statement	Mean Value	Median	Min Value	Max Value	Total Number (*)	"Don't know"	Generally Agree (1+2)	Generally Disagree (3+4)	"Strongly agree" (1)	"Agree" (2)
Assumption 1: The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.										
1.	1.31	1.00	1	2	16	2	100.0%	0.0%	68.8%	31.3%
2.	1.44	1.00	1	2	18	0	100.0%	0.0%	55.6%	44.4%
Assumption 2: The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.										
3.	1.17	1.00	1	2	18	0	100.0%	0.0%	83.3%	16.7%
4.	1.17	1.00	1	2	18	0	100.0%	0.0%	83.3%	16.7%
5.	1.38	1.00	1	2	16	2	100.0%	0.0%	62.5%	37.5%
Assumption 3: Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.										
6.	1.79	2.00	1	3	14	4	92.9%	7.1%	30.8%	69.2%
7.	1.65	2.00	1	2	17	1	100.0%	0.0%	35.3%	64.7%
8.	1.50	1.50	1	2	6	12	100.0%	0.0%	50.0%	50.0%
9.	1.70	2.00	1	2	10	8	100.0%	0.0%	30.0%	70.0%

(*) ... without "Don't Know"
"strongly agree" ... 1
"agree" ... 2
"disagree" ... 3
"strongly disagree" ... 4

10.5.1. *Results of questionnaire and observations*

After the whole study, we analyzed the filled-in questionnaire. The results are presented in Table 6.

We transformed the possible answers into numerical values to empirically analyze them. "Strongly agree" is represented by "1", "agree" by "2", "disagree" by "3", and "strongly disagree" by "4". In the analysis, the "don't know" answers were subtracted out for the following reason. In all cases in which a participant gave the answer "don't know", there was a reason why she could not rate the statement. Thus, it never happened that a participant simply did not have an opinion regarding a statement.

In the first column, references to the statements (S1-S9) of the questionnaire are listed. For each statement, the mean value and median are presented, as well

as the minimum and maximum value. Furthermore, the total number of answers, which are used for calculating the mean value and median, are presented. This number represents the answers without "don't knows", which in turn are summed up and presented in the next column. Finally, we compared the amount of participants who generally agreed with a statement with those who generally disagreed. Finally, we present in the last two columns the difference between participants who strongly agree with those who simply agree with a statement. In addition to the results of the questionnaire, we report about our observations and the feedback from our participants.

All of the participants (68.8 % strongly agreed and 31.3 % agreed) generally agreed with statement 1 that, in contrast to EMF Compare, the Conflict-tolerant Merge incorporates all of their self-performed changes. Furthermore, all participants (55.6 % strongly agreed and 44.4 % agreed) stated that, in contrast to EMF Compare, the Conflict-tolerant Merge incorporates also the changes of the others (cf. statement 2). After the first example was conducted with EMF Compare, most of the participants claimed that their changes are not available in the merged model. In particular, we observed that all participants in the role of Participant A, who have been the first committing their changes, missed them completely. But also the participants in the role of Participant B missed nearly all of her changes. It was often the case that the participants in the roles of Participant B and C did not understand the changes and, especially, the conflicts when using EMF Compare. Thus, they ignored the changes of the others leading to a merged model including only their own changes.

The understandability of the merged model and the occurred conflicts are demonstrated with statements 3 and 4, respectively. All participants generally agreed (83.3 % strongly agreed and 16.7 % agreed) with statement 3 that the merged model of the Conflict-tolerant Merge is more comprehensible than the one of EMF Compare (cf. statement 3). Also the occurred conflicts were more comprehensible when using the Conflict-tolerant Merge (83.3 % strongly agreed and 16.7 % agreed) as demonstrated with statement 4. We observed that all of the participants struggled to comprehend why a conflict occurred when merging with EMF Compare. In addition, for all participants it was difficult to comprehend the intentions behind the parallel changes of the other participants which led to a conflict.

The reason for these results might be explained by statement 5: All participants (62.5 % strongly agreed and 37.5 % agreed) stated that visualizing conflicts in terms of annotations in a merged model instead of visualizing them with the help of correspondences between two versions of a model leads to more understandability. For all participants it was easier to comprehend the conflicts in a merged version of the model although all changes of three participants are merged instead of having two different

versions side by side and then producing a merged version. Further, newly added and also deleted elements are visualized in EMF Compare with correspondences pointing somewhere between two elements in the tree view. This situation led to confusions by nearly all participants, because there did not exist a correspondent model element. Overall, the high understandability of the conflicts and the resulting merged model when using the Conflict-tolerant Merge can be explained by the possibility of visualizing the conflicts in the concrete syntax of the models, in which the participants also performed their changes. In contrast, merging with EMF Compare was impeded by the mismatch of modeling in the concrete syntax and resolving the conflicts in the abstract syntax of the model.

92.9 % of all participants (30.8 % strongly agreed and 69.2 % agreed) generally agreed that their changes are incorporated more adequately in the final model of the Conflict-tolerant Merge than in the one of EMF Compare that was demonstrated with statement 6. However, some of the participants (4 out of 18) stated “don’t know” at statement 6, because when using EMF Compare it often happened that no single change was incorporated in the merged model, because the participants declined the changes of the others when merging due to problems in understanding the changes and resulting conflicts. Thus, it was difficult to assess if a change is “more adequately” incorporated with the Conflict-tolerant Merge.

A similar problem became evident regarding statement 9: Although all participants agreed that all of their changes in the merged version are more correct when using the Conflict-tolerant Merge, 8 participants could not rate this statement, because a comparison with EMF Compare was not possible, when no change was incorporated.

Statement 7 demonstrated that the participants did more agree with the changes of the others incorporated in the final model. All participants agreed (35.3 % strongly agreed and 64.7 % agreed) that the acceptance of the concurrent changes of the others, which led to conflicts, increased with the conflict-tolerant approach, because they better understood the reasons behind the changes.

Finally, to prove that the final models were minimal in the sense that no redundancies existed in the final model, statement 8 was established. However, it did not deliver a result because most of the participants (12 out of 18) could not rate this statement. On the one hand, no redundancies occurred after merging with EMF Compare due to discarding some of the changes by the participants, and, on the other hand, some participants did not recognize the redundancies. Redundancies would occur when merging with EMF Compare, because it duplicates a model element when two developers move it concurrently. However, 6 out of our 18 participants agreed with statement 8 that the changes incorporated in the final model of

the Conflict-tolerant Merge are less redundant than in the one of EMF Compare.

10.5.2. *Implications for assumptions*

Implication for assumption 1 This assumption was that the Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model. Statement 1 and 2 of the questionnaire have been established to check if the merged models are complete. This means that no changes of the participants get lost when merging parallel versions of one model. In the final discussions and according to the results of the questionnaire, all participants agreed that the Conflict-tolerant Merge incorporates their own changes and the changes of the others.

Implication for assumption 2 Statement 3, 4, and 5 were established to find out the understandability of the produced artifacts. According to the observations, interviews, and questionnaire the changes of the participants, the resulting conflicts, and the merged model are more comprehensible when using the Conflict-tolerant Merge. Also the visualization of conflicts in terms of annotations in the merged model are more preferred by the participants of this evaluation. We may state that the Conflict-tolerant Merge makes the reason why conflicts occur more obvious.

Implication for assumption 3 Statement 6 and 7 led to clear results. However, statement 8 could not be answered by several participants as discussed in the subsection before. Thus, we cannot report that the merged model is minimal when using the Conflict-tolerant Merge tool. We also observed that many participants did discovered redundancies in the final model. To test such a criterion, other tests would be more adequate than user experiments as conducted for the evaluation of this approach. Concerning statement 9, 8 out of18 participants could not say whether they agree or disagree with this statement. However, we have also collected qualitative data in terms of observations during the study and interviews after the study. Overall, all participants confirmed that collaborative conflict resolution based on the Conflict-tolerant Merge leads to a final model which is more accepted than merging with EMF Compare following the traditional versioning paradigm.

10.5.3. *Critical discussion*

In this quasi-experimental study, we gained experiences about the usage of the Conflict-tolerant Merge when merging different versions of model in a small team. We conducted a comparison with EMF Compare, but due to the lack of empirical evidence, we do not want to rate both tools. This evidence will be

checked when conducting real-world case studies or controlled experiments in future. In the following, we outline findings of the presented initial evaluation, which will help us to conduct such case studies or experiments.

Within each individual run of the study presented before, three people participated who took different roles with different tasks. For example, it happened that all changes of Participant C were also correctly incorporated with EMF Compare, because she was responsible for the last merge. In contrast, the changes of Participant A and B were at least partly discarded during the merge. This influenced the rating of the statements.

Furthermore, the study was conducted with predefined processes following the conflict-tolerant merge approach and the traditional versioning paradigm. We consider the Conflict-tolerant Merge as basis for collaborative conflict resolution. However, it would also lead to interesting findings when the participants had to resolve the conflicts alone using the Conflict-tolerant Merge. On the other side, EMF Compare is however not designed for resolving the conflicts collaboratively but it would be another interesting variability of the process to let the participants resolve the conflicts together.

We used two different examples for the evaluation to facilitate that one group may evaluate both tools. We designed both examples and the change requests as similar as possible although we used two very different domains to ensure an objective rating of both tools. In addition, for each group we switched example A and B to avoid that the used example influenced the rating of a tool. That means three groups used example A in combination with EMF Compare and, then, example B with the Conflict-tolerant Merge and the other three groups started with example B using EMF Compare.

Furthermore, we prepared relatively small models and not too complex change request for the participants. Another interesting finding would be if the future experiments would lead to the same results when larger models and more complex change requests are used.

As already mentioned, we did not restrict the duration of one individual run, because we wanted to focus on the usability of the systems and the applicability of the Conflict-tolerant Merge approach. In future experiments, it would lead to interesting findings, if the time for merging with EMF Compare is restricted and the quality of the model is then evaluated.

11. Conclusion and future work

In this paper, we presented a novel paradigm for optimistic model versioning. Instead of forcing the modelers to resolve merge conflicts immediately, our system supports deferring the resolution decision until a consolidated decision of the involved parties has been elaborated. Well-defined merge rules are used by our algorithm to incorporate all changes of the modelers, also when overlapping

changes happen, and to mark conflicts by dedicated annotations. The resolution process itself is supported by a conflict resolution model, with which the participants can be assigned to specific conflicts to propose resolutions and find collaboratively a consolidated version. By this method, we transform conflicts into collaborations which results in a higher acceptance of the consolidated model.

With this approach the requirements presented in Section 4 are fulfilled. (1) Modelers may commit their changes without worrying about conflicts and their resolution. They check-in their changes and if conflicts arise they do not have to be resolved immediately. However, by using dedicated annotations, it is ensured that the modifications of the other modelers are not getting lost. (2) Having such an annotated model at hand reflecting all modifications of the modelers, is also beneficial when collaboratively resolving the conflicts and to find together a consolidated version. This approach is based on the assumption that conflicts are not considered as negative results of collaboration, but as chance for indicating improvements. (3) The last requirement was that the versioning system should allow for a better comprehension of the evolution of a model also in cases where conflicts arise. Since we have introduced a lifecycle to the conflicts and since we store the information how a conflict is resolved, modelers can be fully aware of what happened in the development process of a model.

As already discussed in Section 10, the experimental study to evaluate the presented approach was conducted in an artificial context. In addition, interesting findings might arise, when using the approach in a broader industrial context. Real-world case studies would better demonstrate the usefulness and scalability of the approach. However, this would go beyond the scope of this paper and, thus, it is left for future work.

Currently, our implementation supports atomic changes, which may be composed to more complex operations such as refactorings, which are characterized not only by the actual modifications but also by preconditions which have to be fulfilled in order to apply the operation and postconditions which have to hold after a valid application of the operation. Composite changes pose an additional challenge for versioning systems as reported by Dig et al. (2008) and Brosch et al. (2010a) and are subject to future work.

In addition, we are currently enhancing our prototypical implementation to increase its usability. The presented approach demands a novel kind of thinking from the modelers which are used to work with traditional versioning systems. They must be able to work with models containing different viewpoints. In order to avoid distraction by the simultaneously included variants of one model, the user interface design is of particular importance, especially if the models grow big and include many conflicting modifications. Therefore, filters are indispensable to focus on the important aspects and to hide irrelevant details. In addition, strategies and tool support are necessary to guide the consolidation process, e.g.,

by a dedicated conflict browser. As presented in Section 9 the conflict annotation and the additional meta-information are saved as separate model. Conflicts may be integrated in a bug tracking tool or may be send to other users for resolving them, etc.

Notes

1. http://biomedgt.org
2. http://www.omg.org/spec/UML/2.2
3. http://www.bugzilla.org/docs/2.18/html/lifecycle.html
4. http://www.eclipse.org
5. http://www.eclipse.org/modeling/emf/?project=validation#validation
6. http://wiki.eclipse.org/CDO
7. http://www.eclipse.org/emf/compare/
8. http://eclipse.org/modeling/

References

Altmanninger, Kerstin, Martina Seidl, and Manuel Wimmer (2009a). A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, vol. 5, no. 3, pp. 271-304.

Altmanninger, Kerstin, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer (2009b). Why Model Versioning Research is Needed!? An Experience Report. In *MoDSE-MCCM'09. Proceedings of the Models and Evolution Workshop @ MoDELS'09, Denver, Colorado, USA, October 4, 2009*.

Auer, Sören, Sebastian Dietzold, and Thomas Riechert (2006). OntoWiki – A Tool for Social, Semantic Collaboration. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo (eds): *ISWC'06. Proceedings of the 5th International Semantic Web Conference, Athens, Georgia, USA, November 5-9, 2006*, vol. 4273 of *LNCS*. Berlin/Heidelberg, Springer, pp. 736-749.

Balzer, Robert (1989). Tolerating inconsistency. In D. E. Perry (eds): *ISPW'89. Proceedings of the 5th International Software Process Workshop, Kennebunkport, Maine, USA, October, 1989*. Los Alamitos, CA, USA, IEEE Computer Society, pp. 41-42.

Batini, Carlo, Maurizio Lenzerini, and Shamkant B. Navathe (1986). A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, vol. 18, no. 4, pp. 323-364.

Bézivin, Jean (2005). On the Unification Power of Models. *Journal on Software and Systems Modeling*, vol. 4, no. 2, pp. 171-188.

Brosch, Petra, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer (2009). We Can Work It Out: Collaborative Conflict Resolution in Model Versioning. In I. Wagner, H. Tellioglu, E. Balka, and C. Simone (eds): ECSCW' 09. *Proceedings of the European Conference on Computer Supported Cooperative Work, Vienna, Austria. September 7-11*, 2009. Berlin/ Heidelberg, Springer, pp. 207-214.

Brosch, Petra, Gerti Kappel, Martina Seidl, Konrad Wieland, Manuel Wimmer, Horst Kargl, and Philip Langer (2010a). Adaptable Model Versioning in Action. In G. Engels, D. Karagiannis, and H. C. Mayr (eds): *Modellierung'10. Tagungsband der Modellierung 2010, Klagenfurt, Austria, March 24-26, 2010*, vol. 161 of *LNI*. Bonn, GI, pp. 221-236.

Brosch, Petra, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel (2010b). Concurrent Modeling in Early Phases of the Software Development Life Cycle. In G. L.

Kolfschoten, T. Herrmann, and S. Lukosch (eds): *CRIWG'10. Proceedings of the 16th International Conference on Collaboration and Technology, Maastricht, The Netherlands, September 20-23, 2010,* vol. 6257 of *LNCS*. Berlin/Heidelberg, Springer, pp. 129-144.

Brun, Cédric and Alfonso Pierantonio (2008). Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, vol. 9, no. 2, pp. 29-34.

Budinsky, Frank, Dave Steinberg, Ed Merks, Ray Ellersick, and Timothy J. Grose (2003). *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley.

Campbell, Donald T., Julian C. Stanley, and Nathaniel L. Gage (1963). *Experimental and quasi-experimental designs for research*. Houghton Mifflin Boston.

Conradi, Reidar and Bernhard Westfechtel (1998). Version Models for Software Configuration Management. *ACM Computing Surveys*, vol. 30, no. 2, pp. 232-282.

Dey, Anind K. and Gregory D. Abowd (2000). Towards a better understanding of context and context-awareness. In *Proceedings of the Workshop on the What, Who, Where, When and How of Context-Awareness @ CHI'00, The Hague, The Netherlands, April 1-6, 2000*. New York, NY, USA, ACM Press.

Dig, Danny, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen (2008). Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 321-335.

Edwards, W. Keith (1997). Flexible Conflict Detection and Management in Collaborative Applications. In *UIST'97. Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, Banff, Alberta, Canada, October 14-17, 1997*. New York, NY, USA, ACM, pp. 139-148.

Ehrig, Hartmut, Claudia Ermel, and Gabriele Taentzer (2011). A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In D. Giannakopoulou and F. Orejas (eds): *FASE'11. Proceedings of the International Conference on Fundamental Approaches to Software Engineering, Saarbrücken, Germany, March 26 - April 3, 2011*, vol. 6603 of *LNCS*. Berlin/Heidelberg, Springer, pp. 202-216.

Engelbart, Douglas C. and William K. English (1968). A research center for augmenting human intellect. In *AFIPS'68. Proceedings of the Fall Joint Computer Conference, San Francisco, California, December 9-11, 1968*. New York, NY, USA, ACM, pp. 395-410.

Estublier, Jacky, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber (2005). Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, vol. 14, no. 4, pp. 383-430.

Finkelstein, Anthony, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh (1994). Inconsistency Handling in Multperspective Specifications. *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 569-578.

Fitzpatrick, Geraldine, Paul Marshall, and Anthony Phillips (2006). CVS integration with notification and chat: lightweight software team collaboration. In P. J. Hinds and D. Martin (eds): *CSCW'06. Proceedings of the 20th anniversary Conference on Computer Supported Cooperative Work, Banff, Alberta, Canada, November 4-8, 2006*. New York, NY, USA, ACM, pp. 49-58.

Fluegge, Martin (2009). *Entwicklung einer kollaborativen Erweiterung fuer GMF-Editoren auf Basis modellgetriebener und webbasierter Technologien*. Master's thesis, University of Applied Sciences Berlin.

Fowler, Martin (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc.

Fuentes-Fernández, Lidia and Antonio Vallecillo-Moreno (2004). An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional*, vol. 5, no. 2, pp. 5-13.

Gallardo, Jesús, Ana I. Molina, Crescencio Bravo, Miguel A. Redondo, and César A. Collazos (2011). An ontological conceptualization approach for awareness in domain-independent collaborative modeling systems: Application to a model-driven development method. *Expert Systems with Applications*, vol. 38, no. 2, pp. 1099-1118.

Hevner, Alan R., Salvatore T. March, Jinsoo Park, and Sudha Ram (2004). Design Science in Information Systems Research. *MIS Quarterly*, vol. 28, no. 1, pp. 75-105.

Hunter, Anthony and Bashar Nuseibeh (1998). Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 4, pp. 335-367.

Kappel, Gerti, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer (2006). Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio (eds): *MoDELS'06. Proceedings of the International Conference on Model Driven Engineering Languages and Systems, Genova, Italy, October 1-6, 2006*, vol. 4199 of *LNCS*. Berlin/Heidelberg, Springer, pp. 528-542.

Langer, Philip, Konrad Wieland, Manuel Wimmer, and Jordi Cabot (2012). EMF Profiles: A Lightweight Extension Approach for EMF Models. *Journal of Object Technology*, vol. 11, no. 1, pp. 1-29.

Lippe, Ernst and Norbert van Oosterom (1992). Operation-Based Merging. In *SDE 5. Proceeding of the 5th ACM SIGSOFT Symposium on Software Development Environments, Washington, DC, USA, December 9-11, 1992*. ACM, pp. 78-87.

Lukosch, Stephan and Andrea Leisen (2009). Comparing and Merging Versioned Wiki Pages. In J. Cordeiro, S. Hammoudi, and J. Filipe (eds): *WEBIST'08. Proceedings of the 4th International Conference on Web Information Systems and Technologies, Funchal, Madeira, Portugal, May 4-7, 2008*, vol. 18 of *LNBIP*. Berlin/Heidelberg, Springer, pp. 159-173.

March, Salvatore T. and Gerald F. Smith (1995). Design and natural science research on information technology. *Decision Support Systems*, vol. 15, no. 4, pp. 251-266.

Mehra, Akhil, John Grundy, and John Hosking (2005). A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In D. F. Redmiles, T. Ellman, and A. Zisman (eds): *ASE'05. Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, November 7-11, 2005, Long Beach, CA, USA*. ACM, pp. 204-213.

Mens, Tom (2002). A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449-462.

Munson, Jonathan P. and Prasun Dewan (1994). A Flexible Object Merging Framework. In *CSCW'94. Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, October 22-26, 1994, Chapel Hill, North Carolina, USA*. ACM, pp. 231-242.

Niere, Jörg (2004). Visualizing Differences of UML Diagrams With Fujaba. In A. Schürr and A. Zündorf (eds): *Proceedings of the 2nd International Fujaba Days 2004, September 15-17, 2004, Darmstadt, Germany*, vol. tr-ri-04-253. Paderborn, University of Paderborn, pp. 31-35.

Nuseibeh, Bashar, Steve M. Easterbrook, and Alessandra Russo (2001). Making inconsistency respectable in software development. *Journal of Systems and Software*, vol. 58, no. 2, pp. 171-180.

Ohst, Dirk, Michael Welle, and Udo Kelter (2003). Differences between Versions of UML Diagrams. In ESEC'03. *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference, September 1-5, 2003, Helsinki, Finland*. ACM, pp. 227-236.

Renger, Michiel, Gwendolyn L. Kolfschoten, and Gert-Jan de Vreede (2008). Challenges in Collaborative Modeling: A Literature Review. In J. L. G. Dietz, A. Albani, and J. Barjis (eds): *Proceedings of the 5th International Workshop on Groupware: Design, Implementation, and*

Use, @ CRIWG 2009, September 17-19, 2009, Peso da Régua, Douro, Portugal, vol. 10 of *LNBIP*. Berlin/Heidelberg, Springer, pp. 61-77.

Schwanke, Robert W. and Gail E. Kaiser (1988). Living With Inconsistency in Large Systems. In J. F. H. Winkler (eds): SCM'88. *Proceedings of the International Workshop on Software Version and Configuration Control, January 27-29, 1988, Grassau, Germany*, vol. 30 of *Berichte des German Chapter of the ACM*. Teubner, pp. 98-118.

Sebastian, Abraham, Natalya Fridman Noy, Tania Tudorache, and Mark A. Musen (2008). A Generic Ontology for Collaborative Ontology-Development Work-flows. In A. Gangemi and J. Euzenat (eds): *EKWA'06. Proceedings of the 16th International Conference on Knowledge Engineering: Practice and Patterns, September 29 - October 2, 2008. Acitrezza, Italy*, vol. 5268 of *LNCS*. Berlin/Heidelberg, Springer, pp. 318-328.

Shen, Haifeng (2003). *Internet-Based Collaborative Programming Techniques and Environments*. Ph.D. dissertation, Griffith University.

Spanoudakis, George and Andrea Zisman (2001). Inconsistency management in software engineering: Survey and open research issues. In S.-K. Chang (eds): *Handbook of Software Engineering and Knowledge Engineering*. Singapure, World Scientific Publishing, pp. 329-380.

Taentzer, Gabriele, Claudia Ermel, Philip Langer, and Manuel Wimmer (2010). Conflict Detection for Model Versioning Based on Graph Modifications. In H. Ehrig, A. Rensink, G. Rozenberg, and A. Schürr (eds): ICGT'10. *Proceedings of the 5th International Conference on Graph Transformations, September 27 - October 2, 2010, Enschede, The Netherlands*, vol. 6372 of *LNCS*. Berlin/Heidelberg, Springer, pp. 171-186.

Thum, Christian, Michael Schwind, and Martin Schader (2009). SLIM-A Lightweight Environment for Synchronous Collaborative Modeling. In A. Schürr and B. Selic (eds): *MODELS'09. Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, October 4-9, 2009, Denver, CO, USA*, vol. 5795 of *LNCS*. Berlin/Heidelberg, Springer, pp. 137-151.

Tudorache, Tania, Natalya Fridman Noy, Samson W. Tu, and Mark A. Musen (2008). Supporting Collaborative Ontology Development in Protégé. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan (eds): I*SWSWC'08. Proceedings of the 7th International Semantic Web Conference, October 26-30, 2008, Karlsruhe, Germany*, vol. 5318 of *LNCS*. Berlin/Heidelberg, Springer, pp. 17-32.

Wengraf, Tom (2001). *Qualitative research interviewing: biographic narrative and semi-structured methods*. SAGE Publications, London.

Wieland, Konrad, Geraldine Fitzpatrick, Gerti Kappel, Martina Seidl, and Manuel Wimmer (2011). Towards an Understanding of Requirements for Model Versioning Support. *International Journal of People-Oriented Programming*, vol. 1, no. 2. publication pending.

Yin, Robert K. (2002). *Case Study Research: Design and Methods*. SAGE Publications, London.