Concurrent Modeling in Early Phases of the Software Development Life Cycle*

Petra Brosch[1], Philip Langer[2], Martina Seidl[1], Konrad Wieland[1], Manuel Wimmer[1], and Gerti Kappel[1]

[1] Business Informatics Group, Vienna University of Technology, Austria
{brosch,seidl,wieland,wimmer,gerti}@big.tuwien.ac.at
[2] Department of Telecooperation, Johannes Kepler University Linz, Austria
philip.langer@jku.ac.at

Abstract. Software engineering deals with the development of complex software systems which is an inherently team-based task. Therefore, version control support is needed to coordinate the teamwork and to manage parallel modifications. If conflicting modifications occur, in standard approaches the developer who detected the conflict is responsible for the conflict resolution alone and has to resolve the conflict immediately.

Especially in early project phases, when software models are typically employed for brainstorming, analysis, and design purposes, such an approach bears the danger of losing important viewpoints of different stakeholders and domain engineers, resulting in a lower quality of the overall system specification. In this paper, we propose conflict-tolerant model versioning to overcome this problem. Conflicts are marked during the merge phase and are tolerated temporarily in order to resolve them later in a collaborative setting. We illustrate the proposed approach for the standardized modeling language UML and discuss how it can be integrated in current modeling tools and version control systems.

Keywords: team-based modeling, model versioning, conflict tolerance.

1 Introduction

Models are used in nearly all engineering disciplines. Also in software development, model-driven engineering (MDE) recently gained high momentum. In MDE, the powerful abstraction mechanisms of models are not only used for documentation purposes, but also for compiling executable code directly out of models [1]. Like any other software artifacts, models are developed in teams and evolve over time, consequently they also have to be put under version control. Following a pessimistic approach, version control systems (VCS) lock artifacts for exclusive modification whereas in optimistic approaches, VCS support

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584 and the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.

G. Kolfschoten, T. Herrmann, and S. Lukosch (Eds.): CRIWG 2010, LNCS 6257, pp. 129–144, 2010.
© Springer-Verlag Berlin Heidelberg 2010

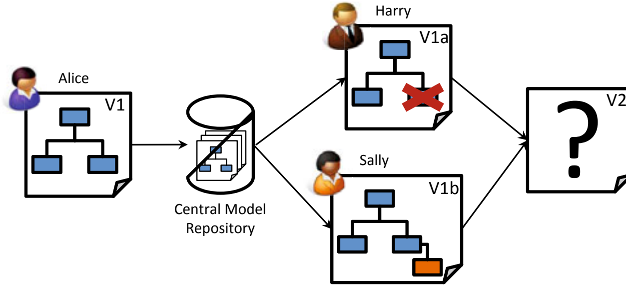

Fig. 1. Optimistic Versioning Process

distributed, parallel team-work. This comes along with the price of merging conflicting changes. Fig. 1 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 standard VCS, Sally is responsable for resolving the conflict immediately. Finally one conflict-free version is checked in with the consequence that some modifications may be 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 and the code has to be executable at any 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 [2]. Models are then used to manage and improve communication among the team members by establishing common domain knowledge. In this context, 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 to use common modeling tools for editing.

This paper is structured as follows. In the next section, we discuss a representative scenario why a collaborative approach to model versioning is needed. In Section 3 we present which kinds of conflicts may occur in model versioning and elaborate on how they are handled in state-of-the-art VCS. In Section 4 our approach of tolerating conflicts when merging different versions of a model is presented. The necessary consolidation phase is discussed in Section 5. Before we conclude with a critical discussion and future work in Section 7, we survey related work in Section 6.

2 Motivating Example

The example depicted in Fig. 2 describes a merge scenario which demonstrates typical problems when developing models in a distributed team following the standard versioning process.

V1. Alice creates a new model in terms of a UML Class Diagram [3] containing the two classes `Person` and `Passport`. She adds the attributes `name` and `bday`—representing the birthday of a person—to the class `Person` and `passNo` to the class `Passport`. Finally, she defines an association between these two classes. 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 passport is always owned by exactly one person. 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 `Person` to `birthday` and adds two new attributes, namely `hash`—a checksum for validating the passport—and `citizen` to the class `Passport`.

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

V1b. In parallel, Sally also understands a passport as a part of a person. Hence, she inlines the attribute `passNo` in the class `Person` and deletes the class `Passport`. Sally commits her changes. The VCS rejects her modification, because they are conflicting with the already performed changes of Harry. A so called "Delete/Update conflict" occurs: the deleted class `Passport` has been changed by another modeler (namely, Harry). Consequently, Sally is responsible to resolve this conflict.

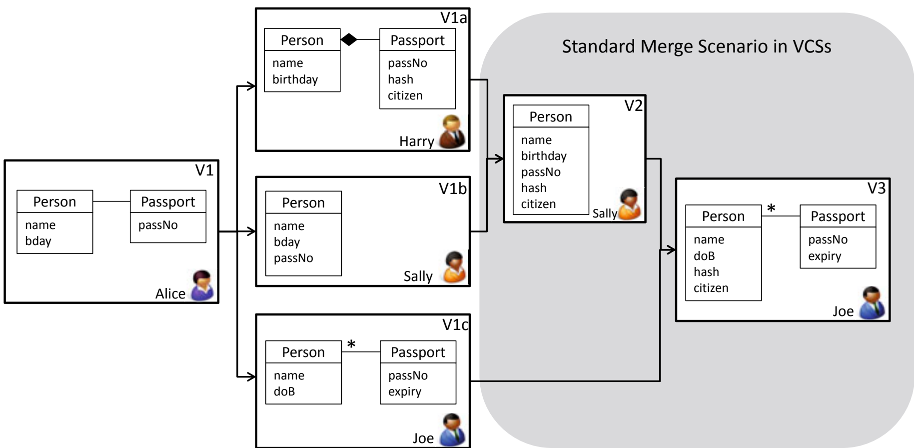

Fig. 2. Motivating Example: Merging in Current VCSs

Resolution. In Sally's opinion, the class `Passport` is still unnecessary in the model, although Harry has updated this class by adding new attributes. Therefore, she decides to inline the added attributes of Harry, namely `hash` and `citizen`, in the class `Person` and to still abandon the class `Passport`. Version V2 now consists of the class `Person` including all attributes added or updated by Alice, Harry, and Sally.

V1c. Joe has also checked out the initial version of Alice (V1) and performs his changes in parallel to Harry and Sally. He renames the attribute `bday` to `doB` in the class `Person` and adds to the class `Passport` a new attribute called `expiry`. Furthermore, Joe is of the opinion that one passport may belong to several persons (i.e., a parent and its children) 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. 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`. He also decides not to delete the class `Passport` as afore propagated by 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 flaws resulting from the conflict resolutions. The final version of the example model does not reflect all intentions of the participating modelers, because there is only one single modeler responsible for the merge of two versions. For this modeler it is difficult to follow the motivation behind the changes of the others, especially, if more than two modelers perform their changes in parallel.

The version V3 of the example described above does not cover the idea of Harry to consider a passport as integral part of one person expressed by an aggregation. In V3 the attributes `hash` and `citizen` have become part of the wrong class, because over the evolution of the model the information that these attributes are referring to the passport get lost.

Note that the shown scenario is only exemplarily and that other merge sequences are possible. In the example, we assume that the modeler who is responsible for the conflict resolution, does her/his best. Currently, only limited tool support is available for such tasks. Usually only one modeler is responsible for the conflict resolution during the merge and it is very likely that this modeler has not the same information as the other modelers which have checked in parallel versions. Consequently, information is lost during conflict resolution.

3 Conflicts in Model Versioning

As we have seen in the previous section, when merging differently evolved versions of one model, various kinds of conflicts might occur [4]. These kinds of

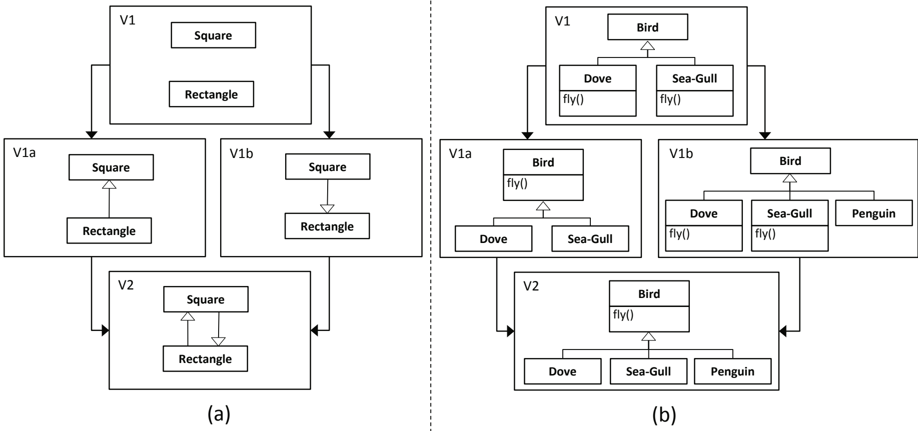

Fig. 3. Example Conflicts: (a) Syntactic Conflict, (b) Semantic Conflict

conflicts are shortly discussed in the following subsection. Furthermore, we elaborate on how state-of-the-art model versioning approaches are handling conflicts.

3.1 Kinds of Conflicts

Overlapping changes like Update/Update or Delete/Update conflicts occur due to concurrent changes on one model element. Such kinds of conflicts are easy to detect. In the motivating example (cf. Fig. 2) two Delete/Update conflicts occur, because Sally deletes the class `Passport` which is modified by Harry and Joe. The example also contains several Update/Update conflicts caused by the parallel modifications of one element, like of the attribute `bday` in the class `Person`.

Syntactic conflicts in a model may be regarded as violation of its metamodel, i.e., the specification of the modeling language. Fig. 3(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.

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. Fig. 3(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 a undesired situation at hand, because a penguin, which is able to fly, contradicts reality.

3.2 State-of-the-Art of Model Versioning

In order to deal with conflicts, according to [5,6], three different approaches are possible. First, techniques could be established to *avoid conflicts* completely. This is accomplished by either versioning in a *pessimistic manner*, i.e., if an artifact is modified, it is locked for all other modelers. Modeling could also be done synchronously, i.e., two modelers are notified immediately, if they are working on the same artifact. Conflict avoidance poses several restrictions on the way how people may work and introduce new complexity in management. This approach does not come along with our goal to develop a model versioning system.

The second approach is to *resolve conflicts* immediately to keep only consistent versions of a model in the repository. Those modelers who are not involved in the conflicting scenario should never notice that conflicts have occurred. In traditional VCS the modeler who is checking in later, bears the full responsibility to resolve the conflicts. This resolution step may be supported with the help of specific rules or policies [7]. Problems have been discussed in the example shown in the previous section.

The third approach is to *tolerate conflicts*. Temporary inconsistencies are accepted in the merged version of a model. The possibly destroyed model has to be repaired by the modelers later and therefore it is taken as basis for discussion leading finally to a model of potentially higher quality. Especially in early phases of software development, such an approach is beneficial, where conflicts could be regarded as possibility to increase the quality of a model through extensive discussions. For the feasibility of this approach, adequate presentation and visualization of the conflicts is indispensable for their resolution. In the remainder of this paper we elaborate on the technical realization of conflict-tolerant versioning.

4 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 of a model. The merge implements our major premiss that neither model elements nor changes get lost and that irrespectively of any occurring conflicts the merge process is never interfered by forcing users to immediately resolve inconsistencies. Consequently, model elements are never truly deleted and conflicts are annotated at the affected model elements. Note that in this paper we only focus on overlapping changes and syntactic conflicts (cf. Sec. 3).

4.1 The Merge Algorithm at a Glance

In this subsection, we present an overview on the conflict-tolerant merge algorithm, which is shown in Algorithm 1. Fig. 4 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

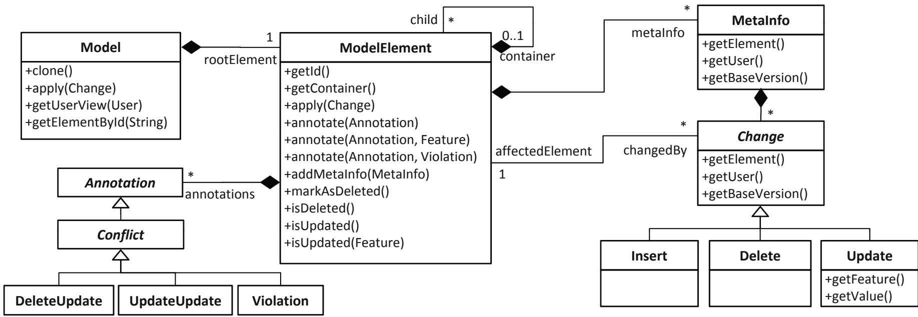

Fig. 4. Types and Operations Used in Algorithm 1

`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 algorithm consists of two phases. In the *fusion* phase (cf. Algorithm 1, line 2 to 28), the algorithm iterates over all changes, checks for conflicts, and applies them to a new merged version irrespectively of conflicts. The algorithm considers three kinds of atomic changes. Model elements may have been *inserted*, *deleted*, or *updated*. A model element is considered as updated, if a specific feature value, e.g., its name, or its container has been changed. According to our premiss, we do not truly remove model elements but only mark them as deleted to retain all model elements and all its historic feature values.

When merging atomic changes, two kinds of conflicts may occur. If a feature of a model element has been concurrently modified by both modelers, an *Update/Update* conflict occurs. A *Delete/Update* conflict appears whenever one modeler deletes a model element which has been updated concurrently by another modeler. These conflicts do not block the fusion phase. We immediately add an annotation to the involved model elements indicating the conflict. For Update/Update conflicts, where feature values are changed in parallel, we set the latest value to the element and save the other value by adding a new meta information object to the modified model element.

In the *validation* phase (cf. Algorithm 1, line 29 to 35), the merged model is validated. Validation means to reveal violations of rules and constraints defined by the modeling language. Because values of model element features might have been updated by multiple users, we first have to derive a view of the model for the specific user who performs the check-in. In this view, the feature values of the specific user are set in the model. Thus, we enable the user to actually validate her changed model incorporating the not directly overlapping changes of other modelers. Otherwise, violations might be indicated that are recently caused by other modelers. Like in the fusion phase, we annotate all model elements subject to violations in this phase. Therefore, we iterate through all revealed violations and annotate the elements involved in the violation.

```
Input: originModel, revisedModel, headModel
Output: mergedModel

   // create a copy of the headModel as base for the new mergedModel
 1 mergedModel = headModel.clone()
   // Fusion: Merging all changes
 2 changeSet = calculateChanges(originModel, revisedModel)
 3 for c ∈ changeSet do
 4   element = mergedModel.getElementById(c.getElement().getId())
 5   if c instanceof Insert then
       // containing element to which new element has been inserted
 6     container = merged-
       Model.getElementById(c.getElement().getContainer().getId())
 7     if container.isDeleted() then
         // container has been marked as deleted
         // -> add DeleteUpdate annotation to container
 8       container.annotate(new DeleteUpdate(c))
 9     end
10     mergedModel.apply(c)
11   else if c instanceof Delete then
12     if
       mergedModel.getElementById(c.getElement().getId()).isUpdated()
       then
         // deleted element has been updated
         // -> add DeleteUpdate annotation
13       element.annotate(new DeleteUpdate(c))
14     end
15     element.markAsDeleted()
16     element.addMetaInfo(createMetaInfo(c))
17   else if c instanceof Update then
18     if element.isDeleted() then
         // updated element has been marked as deleted
         // -> add DeleteUpdate annotation to container
19       element.annotate(new DeleteUpdate(c))
20     end
21     feature = c.getUpdatedFeature()
22     if element.isUpdated(feature) then
23       element.annotate(new UpdateUpdate(c), feature)
24       element.addMetaInfo(createMetaInfo(c))
25     end
26     mergedModel.apply(c)
27   end
28 end
   // Validation: Validate the view of User checking in revisedModel
29 mergedModel_UserView = mergedModel.getUserView(revisedModel.getUser())
30 violations = validate(mergedModel_UserView)
31 for violation ∈ violations do
     // annotate all involved elements violating a model constraint
32   for element ∈ violation.getInvolvedElements() do
33     element.annotate(new Violation(c), violation)
34   end
35 end
```

Algorithm 1. Conflict-tolerant Merge

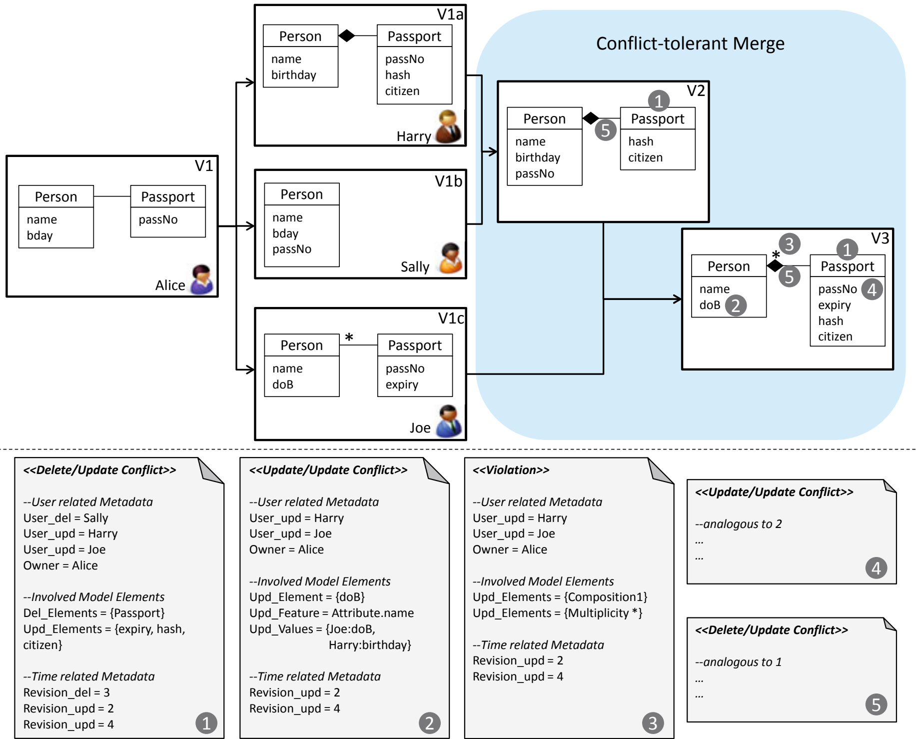

Fig. 5. Conflict-tolerant Merging of the Motivating Example and Annotated Metadata

4.2 Merging the Motivating Example

In this subsection, the afore presented merge algorithm is applied to the motivating example (cf. Fig. 2) in order to illustrate its function in more detail. In contrast to the standard merge depicted in Fig. 2, where conflicts are resolved immediately comparing changed models pairwise, Fig. 5 illustrates the merge results obtained using the *conflict-tolerant merge* and the annotations of the elements marked with the corresponding number.

Merging V1a with V1b into V2. As mentioned afore in Section 2, Sally checks in after Harry has committed his changes to the common repository. His changes comprise two inserts, namely the one of attribute `hash` and of attribute `citizen` in class `Passport`, and two updates. He updated the name of attribute `bday` to `birthday` in class `Person` and changed the aggregation type of the reference connecting `Person` and `Passport` from `unspecified` to `composition`. Consequently, the classes `Passport` and `Person` as well as the reference have to be considered as updated when Sally performs the merge of her version `V1b` with the latest version `V1a` of Harry.

The input for the conflict-tolerant merge algorithm is (i) V1 as the common base revision (originModel in Algorithm 1), (ii) Sally's working copy V1b as version to check in (revisedModel in Algorithm 1), and (iii) Harry's version V1a, which is the most recent version in the repository (headModel in Algorithm 1).

Fusion Phase. As a first step in the merge algorithm, a clone of the most recent version headModel is created (line 1). This new version saved in variable mergedModel acts as basis for incorporating all changes of Sally. Next, the algorithm determines all changes performed by Sally (line 2). This is accomplished by adopting an ID-based model comparison. All changes are saved to changeSet. This set comprises a move of the attribute passNo, which is actually an update of this attribute's container, a deletion of class Passport, and a deletion of the reference connecting Person and Passport. Subsequently, the algorithm iterates over each change in the set. For the first change, i.e., the update of the container relationship of passNo, the variable element takes the value of the attribute passNo in line 4. The current change c is of type Update and the element passNo has not been deleted or updated by Harry, so the change concerning passNo is just applied in line 26 without annotating any conflict. However, for the next change in changeSet, i.e., the deletion of Passport, the change c is an instance of Delete. The element Passport has been changed in headModel by Harry. Thus, the class Passport is annotated with a DeleteUpdate conflict in line 13 and marked as deleted in line 15. Please note that this class has not actually being deleted. In line 16 we create and add meta information to Passport saving informations like who has performed the deletion and which changes are conflicting with that deletion. Like for class Passport, the reference from Person to Passport, which has been updated by Harry and deleted by Sally, is annotated with a DeleteUpdate conflict in the last iteration over the change set.

Validation Phase. Since there were only these three changes performed by Sally, the algorithm proceeds to the validation phase starting in line 29. Now, a user specific view is derived from mergedModel for Sally. This view prioritizes her changes over contradicting changes performed by other modelers while still incorporating all changes that are not overlapping with her changes. Hence, this view contains the class Person without the class Passport because her deletion is prioritized over the update of Harry because she checked in later. However, the independent changes by Harry renaming the attribute bday to birthday and moving the attribute passNo are incorporated because they are not in contradiction with any change of Sally. This view is now validated in line 30 in the algorithm. But since there are no language violations in this model view, the algorithm terminates and the merged version in mergedModel is published into the repository (cf. V2 in Fig. 5).

Merging V2 with V1c into V3. According to Fig. 2, Joe checks in his version V1c, which has to be merged with the head version V2 in the repository. Hence, the input for the merge algorithm is again (i) V1 as the common base version (originModel in Algorithm 1), (ii) Joe's working copy V1c as version to check

in (revisedModel in Algorithm 1), and (iii) V2, which is the most recent version in the repository (headModel in Algorithm 1).

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 Person to Passport. Moreover, he added two attributes to Passport. Sally removed the class Passport and also the reference between Person and Passport.

Fusion Phase. When Joe checks in his version, a clone of the most recent version V2 is created and his changes are calculated and saved in changeSet. This set now contains three changes, namely the update of the name of the attribute bday to doB, the update of the multiplicity of the reference from unspecified to unbound, and the addition of the new attribute expiry in class Passport. In the first iteration over changeSet, the variable element contains the changed attribute bday (line 4) and c is an instance of Update. Hence, the condition in line 17 is fulfilled. Because the element has not been deleted since their common base version V1, no conflict annotations have to be added. In line 21 the feature currently updated by c is determined. Since Joe renamed the attribute, feature points at the name feature of attributes. Unfortunately, the name of this attribute has also been changed by Harry. Thus, an UpdateUpdate annotation for the name feature of the element is added in line 23. Finally, the change is applied. Please note that although the name update of the attribute is applied to the merged version, we do not delete the attribute names "bday" and "birthday". The previous values are saved in the meta information of an element by the apply method. The second iteration concerns the update of the reference multiplicity. Again, the block in line 17 to 27 is executed because c is an instance of Update. Now the updated element, i.e., the reference, has been deleted by Sally. Consequently, a DeleteUpdate annotation is added to the reference. As mentioned before, Harry also updated the same reference. He updated a different feature than Joe did. Joe updated the multiplicity whereas Harry updated the aggregation type. Therefore, no conflict annotations are necessary and the update concerning the multiplicity is just applied to the new merged version. In the final iteration, the variable c contains a change of type Insert. For inserts the variable element is null because there is no element at this point of time in the merged version contained by the variable mergedModel with the ID of the added element. Hence, the container of the added element is fetched in line 6 to check for concurrent deletion. Please note that the container must be available in the model saved in variable *mergedVersion* because the comparison will only report an insert for the added element at the highest position in the containment tree and no more inserts for its implicitly added children. In this case, the container is the class Passport, which has been deleted by Sally. Consequently, a DeleteUpdate conflict is annotated in line 8 and the insert is applied in line 10.

Validation Phase. After all changes have been handled, we may move on to the validation phase. Like before, a user specific view is derived from the model saved in mergedModel. This view corresponds to the model V3 depicted in Fig. 5.

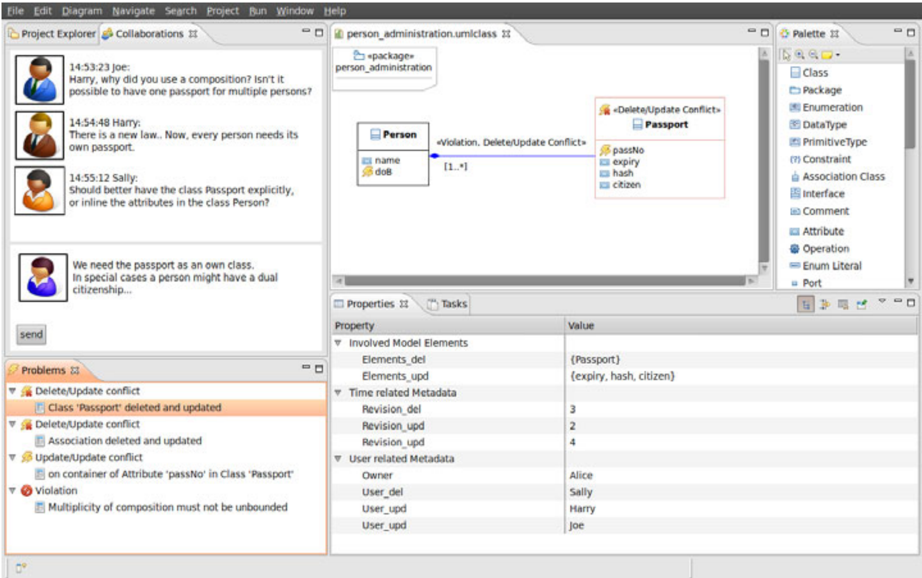

Fig. 6. Screenshot

When this model is validated, a language violation is reported. References of type `composition` may not have an unbound multiplicity (notated as ∗). Consequently, each element involved in this violation is marked. In our example, only the reference is involved and annotated.

5 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 during the software development project, a consolidated model version which reflects a unified view on the modeled domain has to be found by all participants. 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. Fig. 6 depicts a screenshot of the user interface for consideration, which shows all necessary components to resolve the conflicts. On the left side a chat window is provided for discussing the problems to solve. At the bottom left, the conflict list is presented. Each conflict may be selected in order to directly navigate to the involved model elements. Different icons for each type of conflict are used in the conflict report as well as in the model to provide an overview of occurred conflicts. The model itself is displayed and may be edited in the model editor situated in the middle of the window.

By clicking on one of the afore mentioned icons or by selecting a model element in conflict, the metadata describing the conflicting situation is shown in the

property view at the bottom of the window. The metadata created during the conflict-tolerant merge (cf. Section 4) contains information on involved users and model elements of a specific conflict as well as the respective revisions the conflict has been introduced.

Coming back to our running example, Harry, Sally, Joe, and Alice collaborate to resolve all existing conflicts. Supported by the conflict report and the metadata, they find a consolidated version which is depicted in Fig. 7.

Delete/Update: Class Passport, Reference Person to Passport. First of all, they have to decide whether the class `Passport` is needed. Sally did not know that in special cases a person might own several passports of different countries. Since such a situation may only be modeled reasonably with an own class `Passport`, they decide to keep this class. Consequently, they also decide to retain the reference from class `Person` to `Passport`.

Update/Update: Attribute passNo. Now, it is evident how to resolve the Update/Update conflict concerning the container of the attribute `passNo`. They agree to keep `passNo` in class `Passport`.

Violation: Reference cardinality. Harry and Joe are responsible for resolving the metamodel violation, because Harry has introduced the compositional relationship and Joe has set the multiplicity to "unbound". They discuss whether more than one person might be registered in one single passport or if the existence of a passport inherently depends on the existence of the associated person. Finally, they agree to keep the composition without the `unbound` multiplicity.

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`.

After Harry, Sally, Joe, and Alice have finished the resolution of all conflicts, they all accept the final and consolidated version of the model, which is then saved in the repository as new version *V4*.

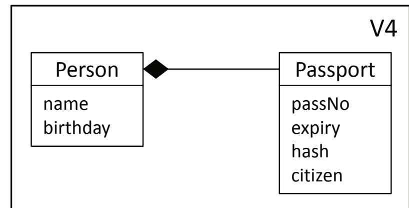

Fig. 7. Consolidated Version

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 is kept and collaboration and discussion is fostered. The resulting final version (cf. Fig. 7) 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.

6 Related Work

With respect to our approach of tolerating conflicts between different model versions, we basically distinguish two areas of related work. First, dedicated systems for versioning software artifacts, and second, approaches dealing with tolerating inconsistencies in software engineering are discussed in the following.

Versioning Systems. In the last decades a lot of research approaches in the domain of software versioning have been published which are profoundly outlined in [8] and [6]. Most of them mainly focus on versioning of source code as they deal with software artifacts in a textual manner. Still, dedicated approaches aiming at the versioning of software models exist. For example, Odyssey-VCS [9] supports the versioning of UML models. EMF Compare [10] is an Eclipse plug-in which is able to match, to compare, and to merge models conforming to the Eclipse Modeling Framework (EMF). CoObRA [11] is integrated in the Fujaba tool suite and logs the changes performed on an artifact. The modifications performed by the modeler who did the later commit are replayed on the updated version of the repository. Conflicts occur if an operation may not be applied due to a violated precondition. Unicase [12] is an Eclipse-based CASE-tool comprising a repository for versioning models. The provided three-way merge technique makes use of editing operations similar to CoObRA.

Although all mentioned versioning systems explicitly support models and different conflict detection mechanisms, they are not aiming at tolerating conflicts. Instead, the conflicts have to be resolved at check-in time by one modeler, only, as it is known from code versioning systems regardless of the development phase.

Tolerating Inconsistencies in Software Engineering. During the late 80ies to the late 90ies, several works have been published which aim at managing inconsistencies (conflicts may be seen as a certain kind of inconsistencies) in the software engineering process. One of the most interesting communalities of these works is that the authors considered inconsistencies not only as negative result of collaborative development, but rather see them as necessary means for identifying aspects of systems which need further analysis or which need to reflect different viewpoints of different stakeholders [13]. 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 [14] have been one of the first who proposed to live with inconsistencies by using a specially adapted programming environment for identifying, tracking, and tolerating consistencies to a certain extent. A similar idea was followed by Balzer [15] 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. [16] presented the ViewPoints framework for multi-perspective development allowing

inconsistencies between different perspectives and their management by employing a logic-based approach which allows powerful reasoning even in cases where inconsistencies occur [17].

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, however, we are strongly focusing on the parallel development of models, thus we have additional conflicts such as update/update or delete/update. 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 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 UML by using a dedicated conflict profile.

7 Critical Discussion and Future Work

In this paper, we proposed 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. This approach is based on the assumption that conflicts are not considered as negative results of collaboration, but as chance for indicating improvements. Only by learning that different views exist, all intentions and requirements are covered in the system under development. This is of particular importance in early phases of the software life cycle when a general agreement has not been established yet. In order to set up a common understanding, i.e., for the specification of requirements, graphical modeling languages like UML are the tool of choice. Models provide a powerful mean to sketch and collect first ideas during brainstorming and design phases. At this time, design flaws are cheap to correct, which might be in later phases not the case anymore.

Due to their graphical representation and inherent extension mechanisms, models may be easily extended with annotations indicating conflicting modifications. These conflicts should be discussed later on within the team and resolved in a consolidation phase when a common agreement has been established. This approach demands a novel kind of thinking from the modelers. They must be able to work with temporarily incorrect 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. Furthermore, strategies and tool support are necessary to guide the consolidation process, e.g., by a dedicated conflict browser.

We are currently enhancing our proof-of-concept implementation to perform extensive case studies—first with students of our model engineering course

(about 150 students) and later in cooperation with our industrial project partner—in order to evaluate the effectiveness of our approach. By this we hope to gain further insights on how people collaborate in early phases of the software development life cycle and how they are handling conflicts in a collaborative setting.

References

1. Bézivin, J.: On the Unification Power of Models. Journal on Software and Systems Modeling 4(2), 171–188 (2005)
2. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman Publishing Co., Inc, Amsterdam (2003)
3. UML Specification 2.2, http://www.omg.org/spec/UML/2.2
4. Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Why Model Versioning Research is Needed!? An Experience Report. In: Joint MoDSE-MCCM 2009 Workshop @ MoDELS 2009 (2009)
5. Edwards, W.K.: Flexible Conflict Detection and Management in Collaborative Applications. In: 10th Annual ACM Symposium on User Interface Software and Technology (UIST 1997), pp. 139–148. ACM, New York (1997)
6. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Trans. on Software Engineering 28(5), 449–462 (2002)
7. Munson, J.P., Dewan, P.: A Flexible Object Merging Framework. In: ACM Conference on Computer Supported Cooperative Work (CSCW 1994), pp. 231–242. ACM, New York (1994)
8. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys 30(2), 232 (1998)
9. Murta, L., Corrêa, C., Prudêncio, J.G., Werner, C.: Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In: 2nd Int. Workshop on Comparison and Versioning of Software Models @ ICSE 2008. ACM, New York (2008)
10. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional (2008)
11. Schneider, C., Zündorf, A., Niere, J.: CoObRA - A Small Step for Development Tools to Collaborative Environments. In: Workshop on Directions in Software Engineering Environments @ ICSE 2004 (2004)
12. Kögel, M.: Towards Software Configuration Management for Unified Models. In: Proc. of the 2nd Int. Workshop on Comparison and Versioning of Software Models @ ICSE 2008. ACM, New York (2008)
13. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Making Inconsistency Respectable in Software Development. Journal of Systems and Software 58(2), 171–180 (2001)
14. Schwanke, R.W., Kaiser, G.E.: Living With Inconsistency in Large Systems. In: Workshop on Software Version and Configuration Control, pp. 98–118 (1988)
15. Balzer, R.: Tolerating Inconsistency. In: 5th Int. Software Process Workshop (ISPW 1989), pp. 41–42. IEEE Computer Society, Los Alamitos (1989)
16. Finkelstein, A., Gabbay, D.M., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency Handling in Multperspective Specifications. IEEE Trans. on Software Engineering 20(8), 569–578 (1994)
17. Hunter, A., Nuseibeh, B.: Managing Inconsistent Specifications: Reasoning, Analysis, and Action. ACM Trans. on Soft. Eng. and Meth. 7(4), 335–367 (1998)