Mining of Model Repositories for Decision Support in Model Versioning*

Petra Brosch**, Martina Seidl, and Manuel Wimmer

Institute of Software Technology and Interactive Systems
Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at

Abstract. State-of-the-art software repositories support optimistic versioning and hence the concurrent editing of one artifact by multiple developers is possible. The drawback of this method is the time-consuming, manual merge process when conflicting changes occur. This is a bigger problem when the artifacts are models. Although similar kinds of conflicts frequently reoccur, current systems hardly provide any resolution support. To tackle this problem, this paper introduces enhanced resolution support based on past resolution decisions. In order to keep the necessary information to learn recommendations to realize improved conflict resolution, a generic extension to current repository technology is proposed.

1 Introduction

Consider the scenario shown in Figure 1: Harry and Sally check out the same artifact from a central repository and perform different changes. When Sally is finished, she loads the new version back to the repository. Later Harry also intends to submit his new version to the repository, but unfortunately his changes are conflicting with the changes of Sally. So he has to resolve these conflicts before he is allowed to store his new version into the repository. The described situation typically occurs when optimistic versioning systems, where—in contrast to pessimistic versioning following the lock-modify-unlock paradigm—the parallel editing of artifacts is possible, are applied for the coordination and management of concurrent team work. If the artifacts under version control are linear like source code, merging parallel changes of multiple developers is a well-developed task [10]. When common version control systems (VCSs) are used for the textual representation of non-linear information, such as analysis and design models, the merge process often becomes time-consuming. The standard line-oriented comparison tools are not suited for the rich-structured models. Since the user support is very limited and consequently most actions are performed by hand, the conflict resolution is often error-prone and cumbersome as well [3].

* This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) and FFG under grant FIT-IT-819584.

** Funding for this research was provided by the fFORTE WIT - Women in Technology Program of the Vienna University of Technology, and the Austrian Federal Ministry of Science and Research.

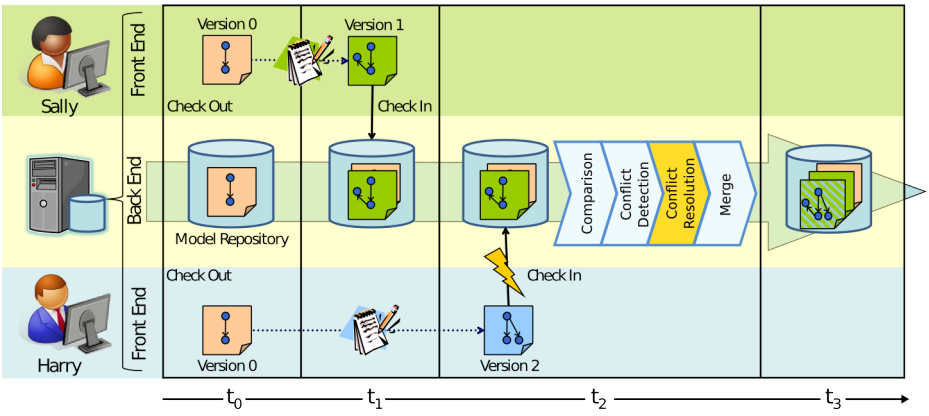

Fig. 1. Check-In Process

To leverage conflict resolution multiple approaches have been proposed [7], but all of them ignore one important factor: the user decisions of the past. As the same kind of conflicts frequently reoccur, the same steps for resolving have to be taken repeatedly. If it would be possible to infer general resolution strategies from accumulated historical data, the time spent for the conflict resolution could be reduced and errors occurring in human intervention could be minimized. The information available in the repository has been successfully used to make predictions about software evolution by applying mining techniques [11]. We intend to follow similar approaches for deriving general resolution strategies, but for this purpose we need data which is not yet kept in standard repositories. Coming back to our previous example, it is impossible to reconstruct the user decisions as, except for explicit branching [9], neither the version of the second check-in of Harry nor the actual conflicts are stored. Therefore we propose an extension for VCSs in order to collect the local working copy of the second modeler, conflicting parts of the model, and former user decisions in order to resolve those conflicts. This allows us an in depth analysis and the mining of general resolution patterns for the realization of a resolution recommender system.

2 Model Versioning By-Example

For further stressing the need for advanced versioning mechanisms, we present three concrete model versioning examples illustrating different kinds of conflicts occurring during the check-in phase.

2.1 Example 1: 4 Merge Strategies—4 Problems

In the example illustrated in Figure 2, two modelers check out the actual version of a UML class diagram (cf. V0 in Figure 2) as their local working copies. Both

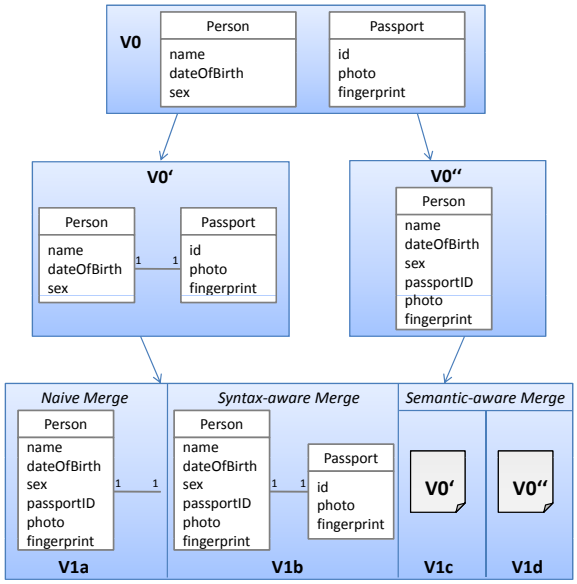

Fig. 2. Example Scenario 1: 4 Ways to Merge Conflicting Changes

perform changes to their copies in order to relate persons with passports. Therefore, the first modeler introduces an association between the classes (cf. V0'), whereas the second modeler shifts all attributes from class *Passport* into class *Person* and afterwards the class *Passport* is deleted (cf. V0''). Based on the conflict detection mechanisms and reasoning power of VCSs [7], the following four merge results (cf. V1a–V1d in Figure 2) are possible.

Naive merge. Employing a naive merge, changes from both modelers are integrated in the merged version without taking care of syntactical or semantical concerns (cf. V1a). The merge incorporates first the added elements into V1a, i.e., the association introduced in V0' and the additional attributes of class *Person* from V0''. Subsequently, the deletions are propagated, i.e., the class *Passport* is no longer available in V1a. The result is a model which is not conform to the metamodel, i.e., the syntax definition of the modeling language, due to a dangling reference problem.

Syntax-aware merge. A more sophisticated merge approach also takes the syntax of the modeling language into account. In our example, the merge may detect that the deletion of the class *Passport* has a dangling reference as consequence. Therefore, the class *Passport* is kept in the resulting model V1b. Note that several other strategies are possible for ensuring syntactic correctness. Although the dangling reference problem is avoided, this model version is also not suitable, because of several redundant attributes.

Semantic-aware merge. For detecting semantic problems such as the redundant attributes, further information is needed for the merge process. A semantic-aware merge may employ heuristics or an ID-based conflict detection to make out that the second modeler has applied a move refactoring for the *Passport* attributes, followed by a rename refactoring for the attribute *ID*, and finally, a delete operation on the class *Passport*. With this input, a merge may conclude that redundant attributes exist when an automatic merge is achieved. Thus, an interactive merge process is started to let the user decide, which of the two versions (`V1c` or `V1d`) is more appropriate for the further development process. Unfortunatly, simply using one version completely ignores model refinements of the second modeler. For more complex examples, also a mix of the two versions or additional modifications are necessary to establish a correct integrated version.

Learning Potential. For this example, manual configuration of the VCSs may be possible by providing dedicated rules for conflict detection and resolution as is done in [8]. But in general, it is not feasible to provide a complete list of all possible conflicts and resolutions in advance. Therefore, the VCS should recognize conflicting model structures and switch to an interactive merge approach, if no automatic conflict resolution is possible. Then the modeler has to inspect and correct the merge result, and the system tracks the user's conflict resolution strategies for self-adaptation.

2.2 Example 2: Refactorings on Models

Changes cause not only *local conflicts* occurring due to concurrent changes of the same model element. In fact, the consequences of the modification of one element may be noticeable over the whole model. Such *global conflicts* often occur when for example refactoring operations are performed.

Consider the following example. One modeler changes class `A` to singleton, i.e., only one instance of this class is created per runtime which is accessed by a public and static method named `getInstance()`. At the same time, a second modeler performs some changes in another class and adds an instance of class `A` by calling the constructor. These changes are conflicting because in the merged version, the constructor is not visible to this class.

Learning Potential. A VCS which detects the changes of converting a class to singleton as refactoring pattern, may apply an appropriate conflict resolution pattern: accept both changes, but change calling the constructor to calling `getInstance()`.

2.3 Example 3: Modeling with Different Intensions

The conflict in our third example (see Figure 3) occurs because of different but partly overlapping intensions of the modelers.

After checking out the actual version of the origin model `V0` consisting of the classes *Car* and *Engine* and the association *has*, Sally replaces the association

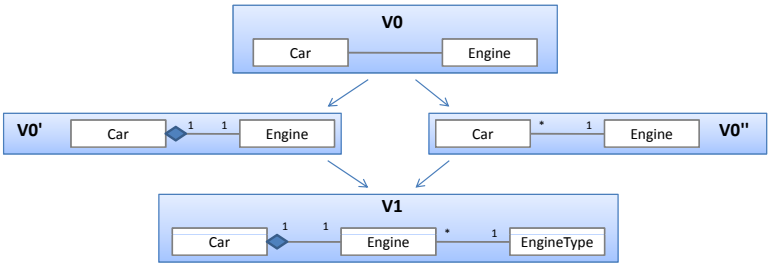

Fig. 3. Example Scenario 3: Different Modeling Intensions

with a composition in her working copy V0'. Hence, she defines an *Engine* instance as part of one *Car* instance. In parallel, Harry increases the multiplicities in his working copy in a different way (cf. Figure 3) to unbound in order to declare that more than one car may use the same type of an engine (e.g., an engine of the type *Diesel*). Both versions express different understandings of the class *Engine*. A naive merge including both variants would result in a semantically incorrect model as the upper bound for the multiplicity of the composition is restricted to one. This leads to a merged model covering both aspects by introducing a third class named *EngineType* and consequently result in a model of higher semantics and quality.

Learning Potential. From this specific conflict and its corresponding resolution, a resolution pattern is mined, namely that in cases where an association is marked as composition and at the same time the multiplicity is set to unbound, an additional class should be introduced. This pattern may be reused for similar examples, such as the following. Consider a model consisting of a class *Library*, a class *Book* and an association between those classes, and concurrently the same modifications as in our running example occur. One modeler defines that the book is contained in one library, actually meaning with book a concrete book copy, whereas the other defines that a book is offered in several libraries. By applying the previously explored resolution pattern, an additional class *BookCopy*—the name has to be inserted by the modelers—is introduced in order to resolve the contradicting association definition.

To address such problems, we present an extension for current VCSs in order to provide enhanced versioning support with a focus on how to provide resolution suggestions.

3 The Conflict Resolution Reasoner

In order to provide decision support for conflict resolution in model versioning, we extend conventional VCSs like Subversion[1] by a learning component, the Con-

[1] http://subversion.tigris.org/

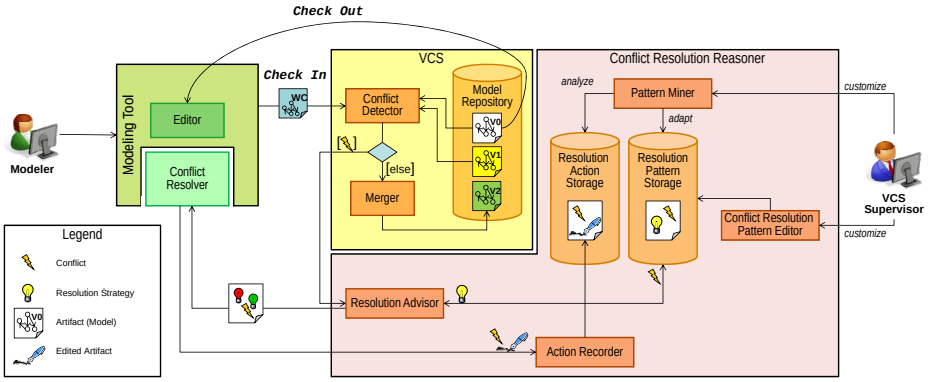

Fig. 4. Architecture of the Extended Model Repository

flict Resolution Reasoner. Furthermore we describe how conventional VCSs may be enhanced to collect additional information for the analysis of past resolution decisions. For the user of the VCS only one thing changes. The system provides recommendations for the resolution which automatically remove the conflicts, if selected. The overall architecture of the extended model repository is depicted in Figure 4.

The main part of the extension is located at the repository server, where resolution actions of all members of the developing team are collected. At the client side, only the diff-tool used for manual conflict resolution and merging has to be replaced.

The workflow is as follows. The client checks out the latest revision of the repository and starts editing. For this step the developer's preferred modeling environment may be used. After editing, the locally modified working copy is committed into the VCS. When a modeler checks in the changes of the working copy, the Conflict Detector compares the working copy with the latest version of the repository—or performs a three-way comparison if the latest version is newer as the original version of the working copy—and decides whether the user has to remodel the working copy, or the merge can be performed without conflicts. In conventional VCSs remodeling is performed completely manually, visualizing textual representations of the user's working copy on the left hand side and of the latest version of the repository on the right hand side. In our approach, the client side Conflict Resolver is not only used for graphical difference visualization and remodeling in order to resolve conflicts, but also for the visualization of resolution recommendations which support the user in deciding a resolution strategy. For providing this decision support, the Resolution Advisor annotates the conflict report produced by the Conflict Detector with appropriate resolution strategies found in the Resolution Pattern Storage.

The Action Recorder collects all user operations performed during the conflict resolution transaction and persists them in the Resolution Action Storage.

The Pattern Miner analyzes the Resolution Action Storage and adapts the Resolution Pattern Storage. In order to find useful patterns the huge amount of data in the Resolution Action Storage has to be preprocessed, meaning that the detailed information about the resolution of concrete conflicts is broken down to more common facts, like it is done in by-example approaches [4]. These facts consist of a set of types of conflicting elements, the context within the model, the type of conflict (e.g., DeleteChange), and a resolution.

The data mining techniques used by the Pattern Miner may be customized by the VCS Supervisor. Starting with computing association rules based on the Apriori algorithm [1] for finding resolution strategies with high confidence for given conflict situations, in future work we plan to use cluster analysis for finding resolution hints in similar situations as well [5, 6].

Resolution patterns found by the Pattern Miner are stored in the Resolution Pattern Storage which acts as the knowledge base for the Resolution Advisor. Already stored patterns may be customized by the VCS Supervisor using the Conflict Resolution Pattern Editor. For example, this may be necessary if an automatically derived pattern is too restrictive in defining the conflict situation in respect of the involved element types.

4 Challenges

We plan to integrate the Conflict Resolution Reasoner in our model versioning system AMOR [2] for offering advanced support in the conflict resolution phase. We are aware of the multiple challenges we have to face for the realization of our vision of building a system which relieves the users of a versioning system from a heavy burden. In the following we discuss some of these challenges and sketch our solution strategies.

Applicability. The benefits of applying the Conflict Resolution Reasoner show up not until the completion of a training phase where user decisions are collected for further analysis. At the beginning, our model versioning system acts like a standard VCS, but it will improve over time. For training purposes, the projects must have a reasonable size. However, for similar projects, i.e., projects using the same modeling language, it is possible to share the discovered knowledge.

Quality of the Resolution Strategies. The resolution of a conflict is a highly sophisticated task which demands human intuition, expert knowledge, and experience. It will probably never be possible to implement a completely automatic conflict resolution tool. Therefore it is crucial to provide guidance and support to the user in the form of a recommender system which proposes multiple resolution strategies from which the user chooses one. The Conflict Resolution Reasoner automatically infers those strategies according to user behavior and user decisions. Naturally, it could happen that useless or even problematic strategies are learnt. Therefore the system is supervised by the VCS Supervisor who may intervene if, for example, the Conflict Resolution Reasoner becomes less reliable by newly learnt strategies. Furthermore the VCS Supervisor may define

conflict patterns and resolution strategies manually. The Resolution Advisor component will rank the strategies considered as suitable according to metrics based on usage statistics.

Performance. We expect a huge amount of data resulting from logging the additional information stored in the Resolution Action Storage. As already explained before, the content of a standard repository does not suffice to reconstruct the occurred conflicts and the accompanying user decisions for resolutions. For the sake of flexibility we decided to use an extra repository with a suitable data structure to store the necessary information for the data mining process. This physical separation has the further advantage that the performance of the part of the backend accessed by the user is not impaired if expensive data processing and complex analysis are performed.

User Acceptance. Since the Conflict Resolution Reasoner is only an additional feature to well-established version control systems and directly integrated in the user's preferred modeling environment almost no familiarization with the tool as well as with the versioning workflow is necessary. Furthermore, if the Conflict Resolution Reasoner does not work—due to whatsoever reason—the user is not hindered at his work because "traditional versioning" is still possible.

Evaluation. When we have implemented our versioning system, we conduct case studies to evaluate the applicability of our approach. Before such a case study, it is almost impossible to test components like the Pattern Miner as (1) there exist no model repositories because models are usually versioned pessimistically and (2) if there was a repository, it would not contain the necessary information anyway. In the context of AMOR, we will evaluate the approach on the one hand with students of a university course and on the other hand in a real-world testbed supported by Sparx Systems[2], the vendor of Enterprise Architect^{TM}.

References

1. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, 1994.
2. K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR - Towards Adaptable Model Versioning. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM'08), Workshop at MODELS'08*, 2008.
3. S. Barrett, P. Chalin, and G. Butler. Model Merging Falls Short of Software Engineering Needs. In *2nd Workshop on Model-Driven Software Evolution (MoDSE 2008), Workshop at CSMR'08*, 2008.
4. P. Brosch, P. Langer, M. Seidl, and M. Wimmer. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM09), Workshop at ICSE'09*, 2009.

[2] http://www.sparxsystems.at/

5. D. J. Cook and L. B. Holder. Graph-Based Data Mining. *IEEE Intelligent Systems*, 15(2):32–41, 2000.
6. U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From Data Mining to Knowledge Discovery in Databases. *AI Magazine*, 17:37–54, 1996.
7. T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.
8. T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. *Model Driven Engineering Languages and Systems*, 4199:200–214, 2006.
9. L. Murta, C. Corrêa, J. G. Prudêncio, and C. Werner. Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models (CVSM08), Workshop at ICSE'08*, pages 25–30, 2008.
10. D. E. Perry, H. P. Siy, and L. G. Votta. Parallel Changes in Large-Scale Software Development: an Observational Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001.
11. T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transaction on Software Engineering*, 31(6):429–445, 2005.