I. Wagner, H. Tellioglu, E. Balka, C. Simone, and L. Ciolfi (eds.).
ECSCW'09: Proceedings of the 11th European Conference on Computer Supported Cooperative Work,
7-11 September 2009, Vienna, Austria

We can work it out: Collaborative Conflict Resolution in Model Versioning[♡]

Petra Brosch[*], Martina Seidl, Konrad Wieland, and Manuel Wimmer
Institute of Software Technology, Vienna University of Technology, Austria
lastname@big.tuwien.ac.at

Philip Langer
Department of Telecooperation, Johannes Kepler University, Austria
philip.langer@jku.at

Abstract. For the versioning of code a pantheon of version control system (VCS) solutions has been realized and is successfully applied in practice. Nevertheless, when it comes to merging two different versions of one artifact, the resolution of conflicts poses a major challenge. In standard systems, the developer who performs the later commit is sole in charge of this often time-consuming, error-prone task. This commit carries the inherent danger of losing the modifications of the other developer. Recently, collaborative merge approaches for code versioning systems have been proposed to minimize this risk. In this paper we propose to apply similar techniques in the context of model versioning where the challenge of merging two versions is even more formidable due to their graph-structure and their rich semantics. In particular, modeling is used in the early phases of the software development, where a collaborative merge is beneficial to elaborate a consolidated understanding of a domain.

Introduction

In this paper we describe an extension of the model versioning system AMOR (cf. Altmanninger et al. (2008)) for collaborative conflict resolution. Following this approach, not only one developer is in charge of merging different versions of

[♡]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.

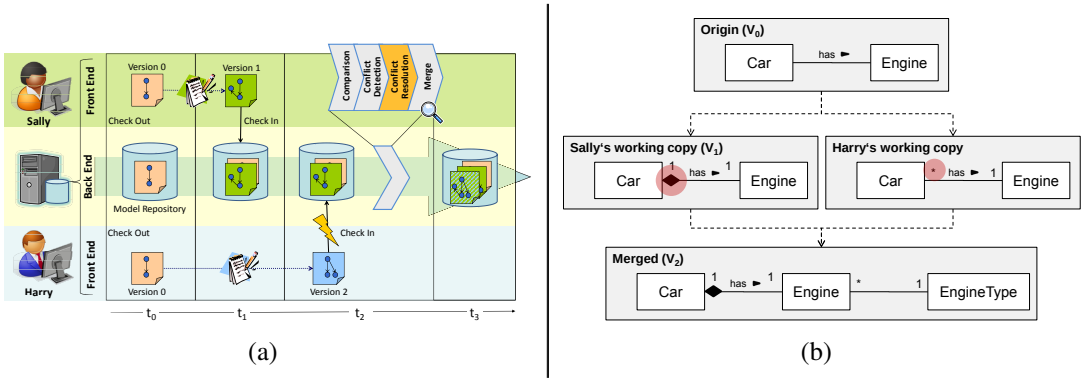

Figure 1. (a) Check-in Process. (b) Motivating Example.

a model but all persons who performed the changes are involved in eliminating the conflicts to obtain one consistent model version.

Collaborative software development without version control systems (VCSs) is nowadays unimaginable. Especially optimistic VCSs are of particular importance because such systems effectively manage concurrent modifications on one artifact performed by multiple developers. In contrast, pessimistic approaches allow only one developer to modify an artifact at the same time. Figure 1(a) illustrates the workflow of applying an optimistic versioning system from the developers' point of view. Two developers (let us call them Harry and Sally) check out the same artifact from a central repository managed by a VCS. When Sally is finished, she loads her new version back to the repository. Later Harry also intends to submit his new version, but the VCS rejects to load his version into the system as his changes are conflicting with the changes of Sally. He is forced to resolve these conflicts before his changes are adopted. The complete responsibility of merging the two versions totally shifts to Harry who may not be aware of Sally's intention and who possibly overwrites or wrongly integrates her modifications. Consequently, the work of Sally could get lost or the merged version could not reflect the intention of Sally. In conventional systems, the consolidation of the other developer is not envisaged. The problems of the asynchronous conflict resolution motivated the augmentation of VCS for code by events notification features as well as with chat possibilities (cf. Fitzpatrick et al. (2006)).

If the artifacts under optimistic version control are text files like source code, merging parallel changes of multiple developers is already a big challenge (cf. Mens (2002)). Merging artifacts as software models is hardly supported by state-of-the-art systems and becomes an even bigger challenge (cf. Barrett et al. (2008); France and Rumpe (2007)). Until now, it was sufficient to version models pessimistically, but as software models nowadays become an indispensable source of information for software engineering—either for documentation purpose or for model driven engineering (MDE) (cf. Bézivin (2005)) where code is automatically generated from models—the need for putting models under optimistic version control arises. Standard VCSs for code usually work on file-level and perform conflict detection by line-oriented text comparison. When applied on the textual serialization of models,

the result is unsatisfactory because the information stemming from the graph-based structure is destroyed and the associated syntactic and semantic information is lost. Consequently, dedicated VCSs for model versioning have been proposed which realize model specific comparison, conflict detection, conflict resolution, and merge components. So far they follow the classical approach where the person performing the later commit of the modified version (i.e., Harry), is left alone with the task of resolving the conflicts. If he has a different understanding of, e.g., the domain, the danger is very high that he destroys the work of the other modeler resulting in unintended models. In this paper we discuss why misunderstandings are even more a potential risk in the context of modeling than in any other area of software development and, inspired by the work of Dewan and Hegde (2007), we propose to integrate a collaborative merging facility into the model versioning system AMOR in order to overcome many problems resulting from the traditional single-person merge process. In particular, modeling is used in the early phases of the software development, where a collaborative merge is beneficial to elaborate a consolidated understanding of a domain.

Motivation

Modeling in parallel and versioning in an optimistic way as already discussed raise several challenges. Concurrent changes of the same model motivated by different but partly overlapping intentions often need manual interventions by the modelers for resolving conflicts. To elaborate on these problems, a concrete example in the context of UML Class Diagram is depicted in Figure 1(b). In this example, 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 with a composition in her working copy V1. 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 1(b)) 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* and therefore an automatic merge is not possible. A naive merge including both variants would lead to a semantically incorrect model as the upper bound for the multiplicity of the composition is restricted to one. A collaborative interaction of both modelers is necessary to find a solution combining both intentions. This exchange of information between the modelers 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. How this collaborative interaction may be integrated in current modeling environments is presented in the next section.

A lot of fertile research effort has been performed to address the challenges of collaborative software development going back to the early nineties (for an overview see Dewan and Riedl (1993)). In a more recent work, Dewan and Hegde (2007) proposed a collaboration model, which enables users to collaboratively resolve conflicts and in further consequence merge their intentions collectively. Still, previous

work only considers the collaborative merge of software code—not software models. An adaption and extension of these concepts in order to match the requirements for collaborative *software modeling* is highly valuable for several reasons.

Software models express aspects of a software system at a much higher level of abstraction and, therefore, reveal a high amount of semantics, domain specific knowledge, and modeling experience. Usually, these skills are distributed variably over all members in a team, which makes the collaboration even more important. The collaboration enables the concentration of these skills—in place and time—where they are needed. Moreover, collaboration is crucial to construct a common vocabulary and understanding. The lack of shared knowledge about goals and concepts of the software often leads to immense non-conformance and low quality in further consequence.

For the sake of minimizing conflicts, development parts are often strictly separated and assigned to team members. Modeling is strongly applied during the requirements engineering phase and analysis phase of a software development project. Since at these early phases the components of the system as well as their borders and interfaces are not clearly defined, the strict separation of tasks is hard to realize. Furthermore, a mistake in early phases leads to disproportionately high costs in later phases. For this reason, it is important to identify design problems as early as possible or even prevent them from the beginning. Design problems in modeling, for example contradicting models, are difficult to detect automatically. The semantic correctness is not expressible in a formal way, so quality assurance mechanisms like unit tests are not applicable. The identification of design problems requires both, modeling as well as domain specific knowledge. Thus, there must be an infrastructure available that supports the joint resolution of conflicts.

Additionally, there exists another requirement for collaborative modeling in contrast to code-centric development: elements as part of a software model may be visualized in more than one diagram. For example, an actor in a UML Use-Case Diagram is often implemented as class within a UML Class Diagram. Both of these two different visual elements (actor and class) represent the same real-world concept. This possibility could recurrently lead to unexpected consequences when concurrent modifications occur and consequently must be considered in conflict detection and resolution.

Finally every member of the team usually feels responsible for his or her work. Using an optimistic versioning approach, coworkers are very often forced to modify a colleague's work in order to resolve a conflict without building a consensus. This frequently leads to semantically inconsistent models, social conflicts and, in the worst case, frustration. Our proposed *collaborative merge* demands to solve such conflicts together, which increases the developers' acceptance as well as the quality of the solution.

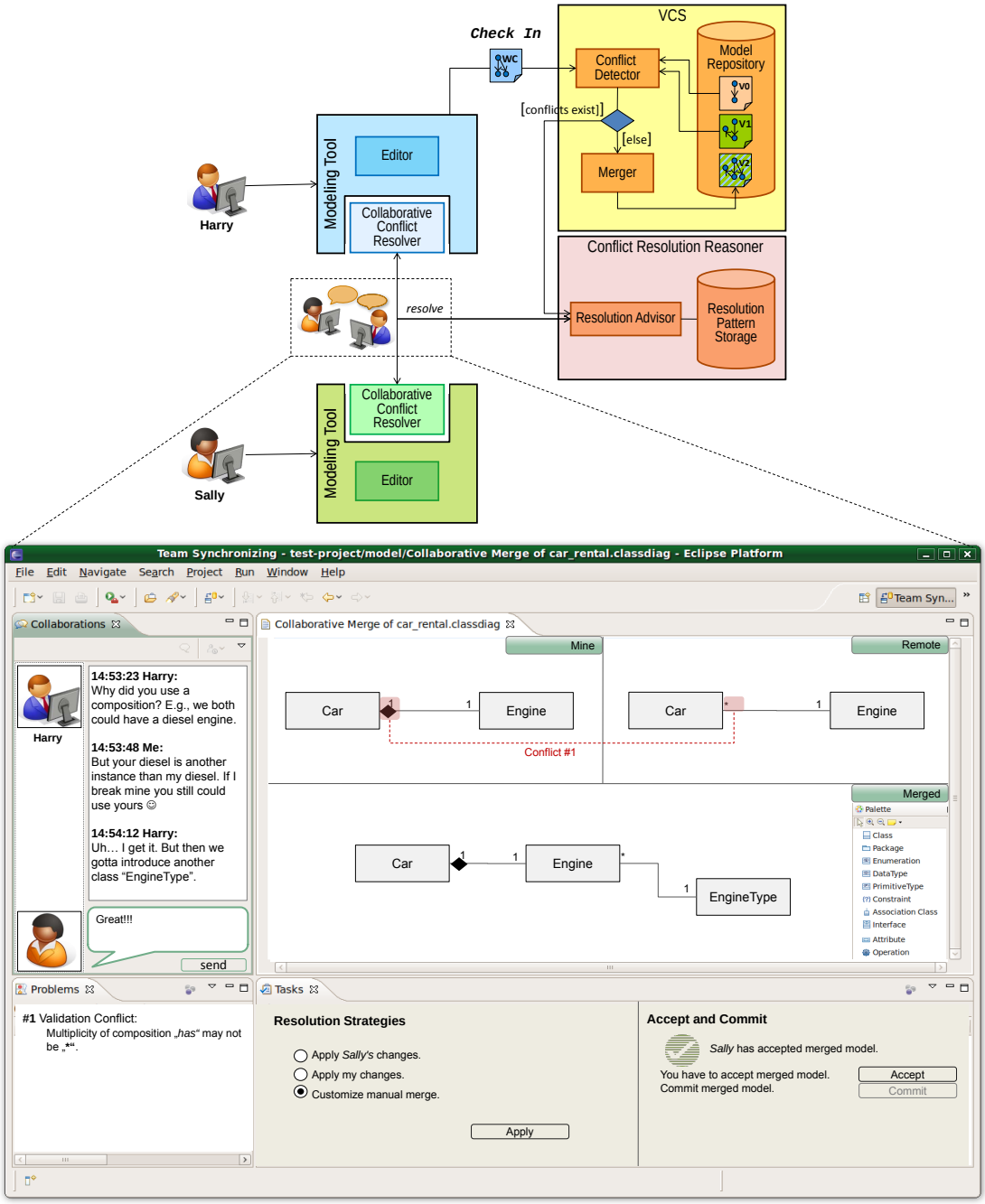

Figure 2. Model Versioning System Architecture and Collaborative Conflict Resolver UI.

Proposed Architecture

In this section, we propose a *semi-collaborative* approach for model versioning. We define semi-collaborative as the parallel editing of artifacts by different modelers is done independently but the task of manual conflict resolution is done in a collaborative manner. In order to overcome the problems mentioned in the previous section, we extend conventional VCSs by a Collaborative Conflict Resolver component to support the collaborative resolution of conflicts between model versions. In addition, to provide recommendations in terms of *resolution patterns* for resolving recurring conflicts, we reuse the Conflict Resolution Reasoner component of the AMOR VCS (cf. Altmanninger et al. (2008)). The creation of resolution patterns is two-folded. First, a VCS administrator defines a first set of resolution patterns in advance which are then recommended in case such conflicts occur. Second, since it is in general not possible to provide resolution patterns for all kinds of conflicts, the Conflict Resolution Reasoner tracks occurred conflicts and their resolution for learning from the user decisions and enhancing the resolution power. For this, the concrete conflict resolutions are broken down to more general facts, formalized in conflict resolution patterns, which are reusable for decision support.

Using the Conflict Resolution Reasoner alone, conflicts are semi-automatically resolved when a resolution pattern is applied by one modeler. While this approach already unifies the resolution knowledge of the whole developing team in terms of decision support, it is still not sufficient due to the following reason. The resolution of conflicting changes is completely in the responsibility of the person who checks in later. This procedure neglects information about the intention of the changes of others. Thus, a person should not be left alone with manual conflict resolution, but rather be accompanied by the author of the conflicting version.

For combining the benefits of both approaches, i.e., providing resolution recommendations and a collaborative merge process, we propose an extended VCS architecture as shown in the upper part of Figure 2. The check-in procedure of an extended VCS is explained by means of the previously presented example. As mentioned before, Harry and Sally check out the newest version V0 and start editing. When Sally is finished, she checks in her working copy and luckily she is the first and no conflicts resulting from concurrent changes occurred. Her working copy is persisted in the repository as V1. Later, when Harry has performed his changes, he checks in his working copy. Unfortunately, the Conflict Detector reports conflicts and an automatically merge can not be performed. Thus, the conflicts must be resolved manually. The Resolution Advisor receives the conflict report and queries the Resolution Pattern Storage for recommended resolution patterns. In our scenario, we assume that three recommendations are available, namely, apply only Sally's changes, apply only Harry's changes, or perform a manual merge. No matter which recommendation is followed, a collaborative merge process is necessary, thus, both authors of the two conflicting versions are informed to resolve the conflict together. Both modelers are notified by their Collaborative Conflict Resolver, which shows the conflict report and possible resolution strategies.

As illustrated in the lower half of Figure 2, the Collaborative Conflict Resolver shows both versions (cf. Mine window and Remote window) where the conflicting elements are marked, as well as a shared whiteboard for collaborative editing the merged version (cf. Merged window). In this window both modelers are able to change the diagram and elaborate a suitable merged model at the same time. Each modification is immediately visualized on the other modeler's screen. The integrated chat component (cf. Collaboration window), the conflict report (cf. Problems window) and the list of resolution strategies (cf. Task window) help discussing the intention of the changes and finding an appropriate solution, respectively. When both modelers are satisfied with the revised merged version, they each press the *Accept* button and finally the *Commit* button is enabled. Then, by clicking on the *Commit* button the revised merged version V2 is stored in the repository, and the applied conflict resolution is propagated back to the Conflict Resolution Reasoner for learning new resolution patterns. In particular, from this specific conflict and its corresponding resolution, a resolution pattern is mined, namely that in cases there 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 examples 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.

Challenges

We integrate the Collaborative Conflict Resolver in the VCS AMOR (cf. Altmanninger et al. (2008)) which provides not only dedicated versioning support for models, but also an advanced conflict resolution component. With the power of these elements, we expect to improve the check-in process in two ways. First, the quality of the merged version increases through the consolidation of all parties involved in modifications. Second, the double manpower for the conflict resolution combined with the solution strategies offered by the Conflict Resolution Reasoner of AMOR may decrease the time spent on model merging. We are aware of the multiple challenges on the way to the realization and practical application. In the following, we shortly discuss some of our concerns and outline solution statements.

User Acceptance. The success of our component highly depends on the willingness of modelers to communicate with each other and to engage with unfamiliar elements newly introduced in the versioning process. We expect a high acceptance by the users as soon as the advantages become evident, i.e., they save time and can focus on their own work and not on the work of others.

Visualization. Strongly connected to the user acceptance is the provision of an attractive user interface. Of paramount importance is the well-arranged presentation of the very diverging information. In the previous section we have already presented a prototypical depiction of the Collaborative Conflict Resolver, but we aware that if our system should be used in the large scale, much effort will have to be put in the evaluation and the resulting suggestions.

Scalability. In the previous scenario only two modelers were involved in the conflict situation. In practice, even more persons will work on one model and quite naturally, the following question arises. How does the system behave if the modification of more than two developers are conflicting. In practice, this does not yield the problem, as it is possible to trace back the situation to the base case of only two conflicting versions, because a working copy only has to be merged with the actual version in the repository.

Evaluation. We plan to conduct case studies where we evaluate the fertility of our approach. As our system highly depends on user interaction, tests which are not performed in the context of a real world project of reasonable size (more than one modeler) are not expressive. We have the opportunity to assess the viability of our new component with students of a university course as well as in the context of a real world test bed provided by our industrial partner Sparx Systems, the vendor of Enterprise Architect™.

References

Altmanninger, K., G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer (2008): 'AMOR—Towards Adaptable Model Versioning'. In: *Proc. of the 1st International Workshop on Model Co-Evolution and Consistency Management*.

Barrett, S., P. Chalin, and G. Butler (2008): 'Model Merging Falls Short of Software Engineering Needs'. In: *Proc. of the 2nd Workshop on Model-Driven Software Evolution*.

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

Dewan, P. and R. Hegde (2007): 'Semi-synchronous conflict detection and resolution in asynchronous software development'. In: *Proc. of the 10th European Conference on Computer-Supported Cooperative Work*. pp. 159–178, Springer.

Dewan, P. and J. Riedl (1993): 'Toward Computer-Supported Concurrent Software Engineering'. *IEEE Computer*, vol. 26, no. 1, pp. 17–27.

Fitzpatrick, G., P. Marshall, and A. Phillips (2006): 'CVS integration with notification and chat: lightweight software team collaboration'. In: *Proc. of the 2006 ACM Conference on Computer Supported Cooperative Work*. pp. 49–58, ACM.

France, R. B. and B. Rumpe (2007): 'Model-driven Development of Complex Software: A Research Roadmap'. In: *Proc. of the 29th International Conference on Software Engineering*. pp. 37–54, IEEE Computer Society.

Mens, T. (2002): 'A State-of-the-Art Survey on Software Merging'. *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 449–462.