Adaptable Model Versioning in Action[1]

Petra Brosch[2], Gerti Kappel[3], Martina Seidl[4], Konrad Wieland[5], Manuel Wimmer[6], Horst Kargl[7] and Philip Langer[8]

Abstract: In optimistic versioning, multiple developers are allowed to modify an artifact at the same time. On the one hand this approach increases productivity as the development process is never stalled due to locks on an artifact. On the other hand conflicts may arise when it comes to merging the different modifications into one consolidated version. In general, the resolution of such conflicts is not only cumbersome, but also error-prone. Especially if the artifacts under version control are models, little support is provided by standard versioning systems.

In this paper we present the enhanced versioning process of the model versioning system AMOR. We show how AMOR is configured in order to obtain a precise conflict report which allows the recommendation of automatically executable resolution patterns. The user of AMOR chooses either one of the recommendations or performs manual resolution. The manual resolution may be in collaboration with other developers and allows to infer new resolution patterns which may be applied in similar situations.

1 Introduction

The development of software systems 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 at the same time. Like other software artifacts, models are developed in teams and evolve over time; consequently they also have to be put under version control.

Standard VCSs for code usually perform the conflict detection by line-oriented text comparison of arbitrary artifacts. When applied on the textual serialization of models, the result is unsatisfactory because of the graph-based structure, single changes on the model may result in multiple changed lines in the textual serialization. Considering lines as unit of comparison, the information stemming from the graph-based structure is destroyed and

[1] This work has been partly funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584 and by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research.

[2] brosch@big.tuwien.ac.at
[3] kappel@big.tuwien.ac.at
[4] seidl@big.tuwien.ac.at
[5] wieland@big.tuwien.ac.at
[6] wimmer@big.tuwien.ac.at
[7] horst.kargl@sparxsystems.at
[8] philip.langer@jku.at

associated syntactic and semantic information is lost. Consequently, dedicated VCSs for models have been proposed (cf. [Alt08, KGE09, OWK03, MCPW08, CRP08, SZN04, OS06, Kög08, AP05]). However, they either support generic model versioning and therefrom do not consider language-specific aspects or they are built for a specific language and are not suitable for other languages. Furthermore, the focus in such systems is mainly set on the detection of conflicts. The actual resolution is usually left to the user without hardly any dedicated support. The provided resolution facilities are mostly limited to refuse modifications or to perform a manual remodeling. This task is often very repetitive as the same conflict types occur again and again. These systems are not capable to analyze the user's behavior and learn therefrom for similar situations. Finally, the merge is left to the person who does the later check-in. Therefore it is totally her responsibility to obtain a consolidated version containing all modifications in a high quality. Collaborative approaches as for code versioning [DH07], where all responsible developers are involved in the merge process, have not been realized in model versioning systems so far. Consequently, the different intentions of the modelers might not be captured in the merged version and might get lost. As models are often applied for early project phases, where a common understanding and a common terminology might not have been established yet, this loss could cause errors which become obvious only in later phases of the project and which then are very hard and expensive to correct.

The adaptable model versioning system AMOR [AKK+08] aims to combine the advantages of both generic and language-specific VCSs by providing a generic framework with extension points for including language-specific features. AMOR is built around subversion[8] and attaches an extended version of EMF Compare[9] for change detection. Hence, AMOR can deal with arbitrary Ecore[10] based modeling languages. The combination of generic and specific aspects improves conflict detection as well as conflict resolution. When manual resolution is necessary, a collaborative merge process might be initiated. If the resolution is performed manually, it is analyzed in order to derive resolution recommendations for similar situations which occur in future scenarios. In this paper we present the conflict detection and conflict resolution components of AMOR from the workflow perspective and show how their interplay eases the check-in process for models.

The remainder of this paper is structured as follows: In Section 2 we introduce a motivating conflict scenario in which a naive merge would change the originally intended semantics. In Section 3 we show how such conflicts are detected and resolved within the model versioning system AMOR and in Section 4 we present a novel approach for the automatic derivation of a general resolution strategy for similar conflicts. Section 5 gives an overview on related work. Finally, in Section 6, we draw conclusions and sketch some future work.

[8] http://subversion.tigris.org
[9] http://wiki.eclipse.org/index.php/EMF_Compare
[10] http://www.eclipse.org/modeling/emf

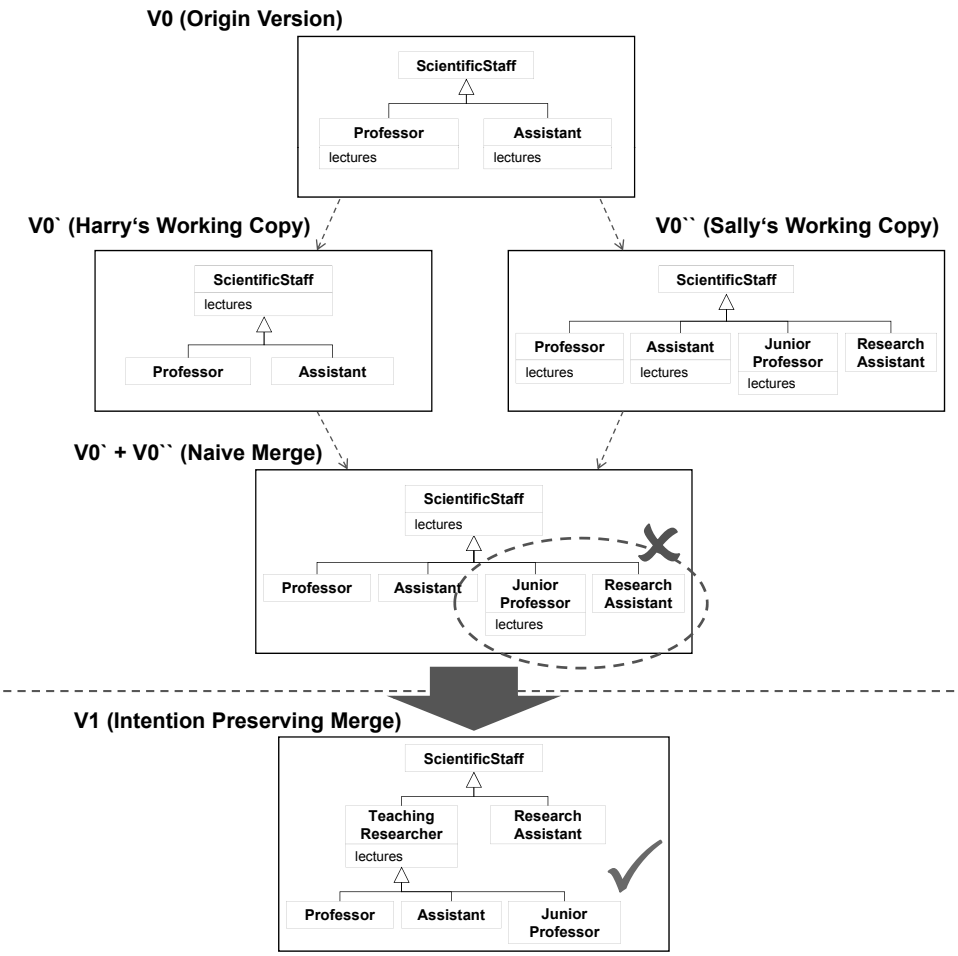

Fig. 1: Conflict Scenario.

2 Motivating Example

In the following, we introduce a small use case in which two modelers, Harry and Sally, concurrently modify the common origin model. In this scenario a naive merge of both working copies leads to an unintended effect regarding the semantics of the merged model.

Conflict Scenario. Both, Harry and Sally work on the common base model V0 depicted in Figure 1. The model contains the two classes Professor and Assistant which are subclasses of ScientificStaff. Each of them contains a property lectures. Harry decides to perform the refactoring “Pull Up Field” [FBB+99] by shifting the property lectures to the superclass ScientificStaff. His modifications are depicted in version V0’ of Figure 1. In the meanwhile, Sally introduces a new subclass JuniorProfessor which includes the

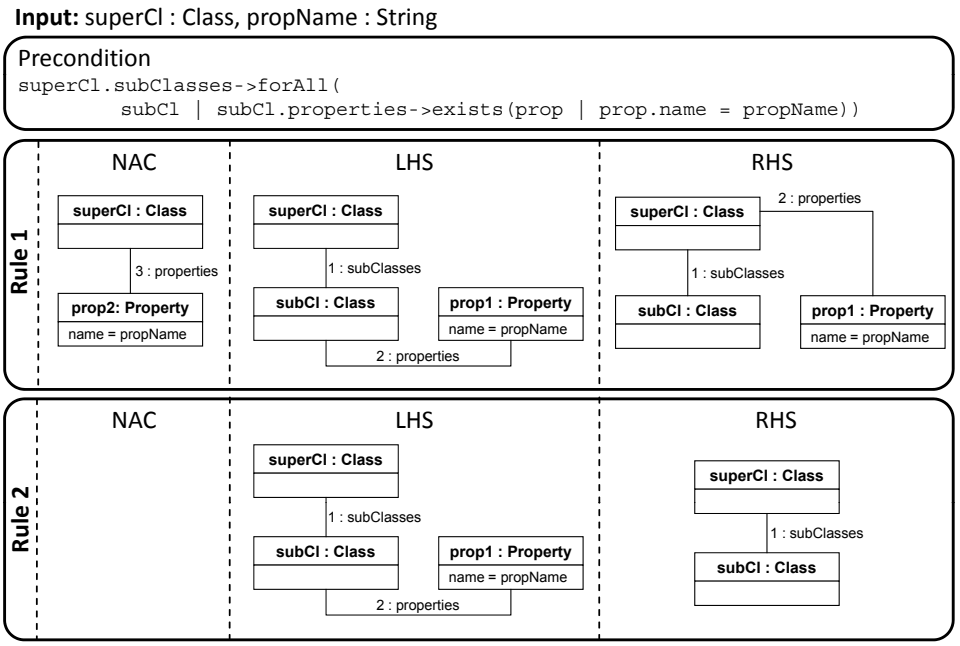

Fig. 2: Graph Transformation Rules for "Pull Up Field".

property lectures like the other existing subclasses. Additionally, she adds a new subclass ResearchAssistant which does not contain the field lectures (cf. V0" in Figure 1) since research assistants (at least in theory) are not holding lectures.

When merging naively there are no conflicts, because no overlapping changes are at hand. The modifications of both modelers are incorporated in the merged version V0' + V0" depicted in Figure 1. However, a closer look at the merged version reveals two problems. The first obvious one is due to the redundantly declared field lectures in class JuniorProfessor which already inherits the aforementioned field from ScientificStaff. Secondly, the class ResearchAssistant now also inherits the field lectures. This has not been intended by Sally who has introduced this class in V0". Thus, the merge has changed the originally intended semantics of the model, which should not happen when refactorings are applied.

Intention Preserving Merge. The aforementioned issues are only detectable if the applied changes are correctly identified. Especially, Harry's refactoring "Pull Up Field", a composite operation, has to be known to the conflict detection component in order to indicate the conflicts.

In AMOR language specific composite operations may be defined by the user to improve change detection. Figure 2 shows such a composite operation represented by a graph transformation rule. The refactoring "Pull Up Field" is defined by depicting required inputs, preconditions, and transformation rules. This refactoring needs two inputs—the superclass

to which the field is shifted and the name of the field to be pulled up. The depicted precondition ensures the preservation of the model's semantics by stipulating all subclasses to contain a property with the specified name. Consequently, the refactoring may only be applied to a superclass superCl if all of its subclasses provide a property with the name propName. Rule 1 is executed if superCl does not yet contain the field to be pulled up. In this rule the property, currently contained by a subclass, is shifted to the superclass. Due to the negative application condition this rule is executed only once. Henceforward, Rule 2 removes the shifted field from the subclass as long as there are subclasses containing the field to be pulled up.

With the knowledge about the applied refactoring, the introduction of the redundant field lectures can be avoided by first applying atomic operations like "Add Inheritance" and then replaying the composite operation, "Pull Up Field". The newly introduced elements, i.e., the JuniorProfessor, are then included in the refactoring. However, reconsidering the conflict scenario in Figure 1, Harry's refactoring cannot be directly applied to Sally's working copy. The subclass ResearchAssistant does not contain the field to be pulled up and therefore violates the refactoring's precondition.

The optimal merge, which preserves all intentions of both modelers, consists of the introduction of a new class TeachingStaff containing the property lectures as subclass of ScientificStaff (cf. V1 in Figure 1). The classes Professor, Assistant, and JuniorAssistant are subclasses of the newly created class TeachingStaff and consequently inherit the property lectures. In contrast, the class ResearchAssistant is a direct subclass of ScientificStaff without the property lectures. Hence, instances of ResearchAssistant cannot have lectures. So the intentions of Harry (optimization) and Sally (extension of the model) are adequately captured.

3 The AMOR Model Versioning System

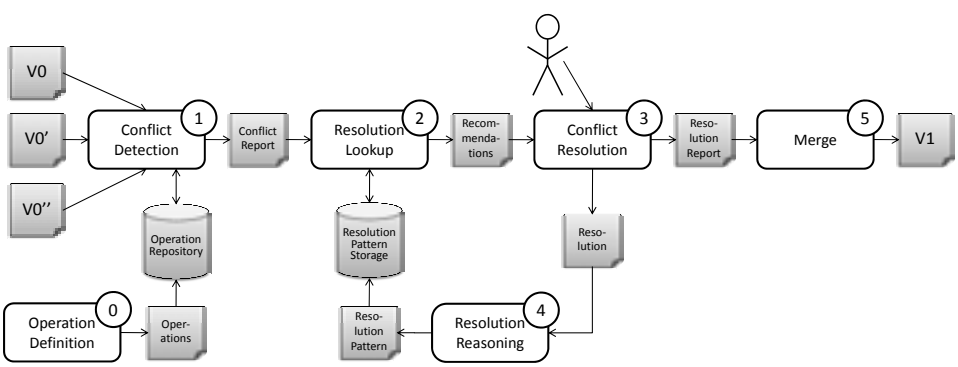

Fig. 3: AMOR Workflow.

In the following, we show how the model versioning system AMOR supports Sally to correctly merge the different versions of the origin model. Therefore, we first give an overview how AMOR extends the workflow found in common versioning systems and then we present how the different components of AMOR operate.

3.1 The AMOR Workflow

In the following, we employ the afore presented example and accompany Sally doing the check-in of her modifications and resolving the afore described conflict with AMOR. The model versioning system AMOR provides not only an enhanced conflict detection component, but also offers user support even to conflicts resulting from the application of composite operations like refactorings. The complete workflow is depicted in Figure 3.

0. *Operation Definition.* The model versioning system AMOR provides modeling language-independent versioning support. The quality of the detection and resolution of conflicts may be considerably improved when language-specific knowledge is incorporated in the merge process [DJ06]. Therefore, AMOR provides an extension point to integrate composite, user-defined operations like refactorings. Once this is done, applications of such refactorings are detectable. This additional information is the basis for a precise conflict detection.

1. *Conflict Detection.* For the comparison and conflict detection the inputs are the origin version V0 and Harry's and Sally's working copies V0' and V0". The conflict detection component detects not only the generic atomic changes like *insert*, *update*, *delete*, but also composite operations stored in the *Operation Repository*. In our example, a conflict between the operations "Pull Up Field" and "Add Inheritance" is reported. If an additional class is added as subclass of ScientificStaff and this new subclass does not contain the attribute lectures, the refactoring "Pull Up Field" is not applicable any more.

2. *Resolution Lookup.* In this step, the *Resolution Recommender* of AMOR checks whether there are solutions for the reported conflict in the *Resolution Pattern Storage*. For the moment, it is assumed that nothing suitable is found. Consequently, the only recommendations which are proposed to Sally are either to apply Harry's refactoring omitting her own changes or to apply her own changes and excluding Harry's refactoring.

3. *Conflict Resolution.* Sally has to decide how to resolve the conflict. She may either resolve the conflict completely manually or choose one of the recommendations made by the *Resolution Recommender*. As mentioned before, there are no useful recommendations and therefore Sally chooses to apply a manual resolution. Since Sally is not sure about Harry's intention, she decides to perform a collaborative resolution where she consolidates Harry. Their solution contains the modifications of both where the new class TeachingStaff is introduced as shown in Figure 1.

4. *Resolution Reasoning.* In the background, the *Resolution Reasoner* analyzes Harry's and Sally's decisions in order to derive a general resolution pattern for the conflict between the "Pull Up Field" and "Add Inheritance" operations. The derived resolution introduces a new intermediate class containing the pulled up property and positions this class at the appropriate place in the inheritance hierarchy. The pattern is stored in the *Resolution Pattern Storage* for the application in similar situations.

5. *Merge.* Finally, all previously chosen resolution recommendations are applied and the resulting model is saved into the repository as a new version.

3.2 The Components of AMOR

Operation Definition. In step 1 the merge process comprises the detection of conflicts concurrently performed by the modelers. Beside atomic changes like *insert*, *update*, and *delete*, modelers may also have performed composite operations like refactorings. Detecting and regarding these composite operations enhances the preservation of both modelers' intentions, as reported in [DJ06]. Of course, composite operations are always specific to a certain modeling language. The AMOR conflict detection component may be enhanced with user-defined composite operation specifications, which describe composite operations in terms of a set of atomic operations and necessary pre- and postconditions. AMOR provides a tool called *Operation Recorder* to easily specify composite operations for specific languages by example [BLS+09b]. These *Operation Specifications* are then either interpreted by the AMOR system or used to derive executable representations like graph transformations.

Conflict Detection. Once an operation specification is created and included in the *Operation Repository*, the *Change Detector* is able to identify applications of the respective composite operation. The detection mechanism is implemented by searching for the operation pattern contained in the *Operation Specification*. If the pattern is found and the model elements referenced by the matching operations fulfill the pre- and postconditions, an application of the composite operation is at hand. This detection allows a more compact representation of the difference and conflict reports by folding atomic operations which belong to a composite operation.

Based on the applied changes, conflicts are easily detected if the same element is modified in different versions of a model. Conflicts resulting from changes of different elements are much harder to detect. Especially, if composite operations are involved, standard versioning systems do not reveal conflicts and merge problems related to the application of the composite operation. To overcome this drawback, AMOR creates a tentative merge by first applying all non-interfering atomic changes and subsequently by replaying the executed composite operations to the common base version. Consequently, added or changed model elements are enclosed in the reapplied composite operation. On the one hand, this maximizes the combination of the original modelers' intentions and, on the other hand, reveals inconsistencies concerning the compatibility of operations. For instance, such inconsistencies occur if a composite operation cannot anymore be executed to the model after all atomic changes are applied. This is accomplished by testing the preconditions of the respective composite operation in the tentatively merged model.

Resolution Lookup. Recent VCSs indicate where conflicts interfere the merge process, but they hardly provide any resolution support to the user. AMOR provides the *Resolution Recommender* which offers resolution recommendations to the modeler who is responsible to perform the change correctly. These recommendations are stored in the *Resolution*

Pattern Storage. A resolution recommendation is a pair containing a conflict description and an executable operation pattern. The *Resolution Recommender* matches the conflicts obtained from the conflict report against the conflict descriptions in the *Resolution Pattern Storage*. The matching resolution recommendations are presented to the user ordered according to their relevance. The *Resolution Recommender* identifies not only perfectly matching resolution recommendations, but also proposes resolution recommendations for similar conflict situations by abstracting the resolution pattern (see Section 4).

Conflict Resolution. AMOR provides different mechanisms to support the conflict resolution which are described in the following.

- *Semi-automatic vs. Manual Conflict Resolution.* In semi-automatic conflict resolution the user selects one of the suggested recommendations offered by the *Resolution Recommender*. The resolution rule of the resolution recommendation is then applied on the maximally merged version. It is executed completely automatically as described in [BLS+09b]. In certain cases user input is required, e.g., if the name of an element has to be introduced. If none of the given suggestions fit the user's requirement, then it is still possible to resolve the conflict manually either alone or in collaboration with other modelers. This process is described in the next paragraph.
- *Collaborative vs. Single User Conflict Resolution.* When no adequate recommendations are found, then semi-automatic resolution is not possible. If one single modeler decides how to merge, the danger is inherently high that the merged version does not reflect the intentions of all involved modelers. Therefore, AMOR offers on the one hand a validation of the merged version and on the other hand an opportunity to resolve conflicts in a collaborative way. In this context, collaboration refers to communicating the intentions behind the changes and trying to combine these intentions in a new version. Thus, AMOR provides an extension called *Collaborative Conflict Resolver* [BLS+09a] to overcome the before mentioned issues by orchestrating the modelers when resolving conflicts. The *Collaborative Conflict Resolver* offers the modelers a communication platform to exchange the intentions behind their conflicting changes. The modelers may manually create a merged version together by partly remodeling the scenario to reach a model of high quality. Afterwards, both modelers have to accept the new version before a commit to the central model repository is possible. This ensures an approved and consolidated version to be checked in. To sum up, the *Collaborative Conflict Resolver* allows to distribute the responsibility in this critical and error-prone merge phase. Solving conflicts in collaboration ensures the preservation of all intentions of the modelers and, overall, increases the acceptance of the merged version and the whole development process.

Resolution Reasoning. The mission of the *Resolution Reasoner* is twofold. On the one hand, the *Resolution Reasoner* tries to derive resolution patterns if the solution is obtained manually, which can be later used for automatic resolution. This is done by reusing mechanisms of the *Operation Recorder*. On the other hand, the *Resolution Reasoner* tracks the application of already existing resolution patterns and persists metadata for ranking purposes.

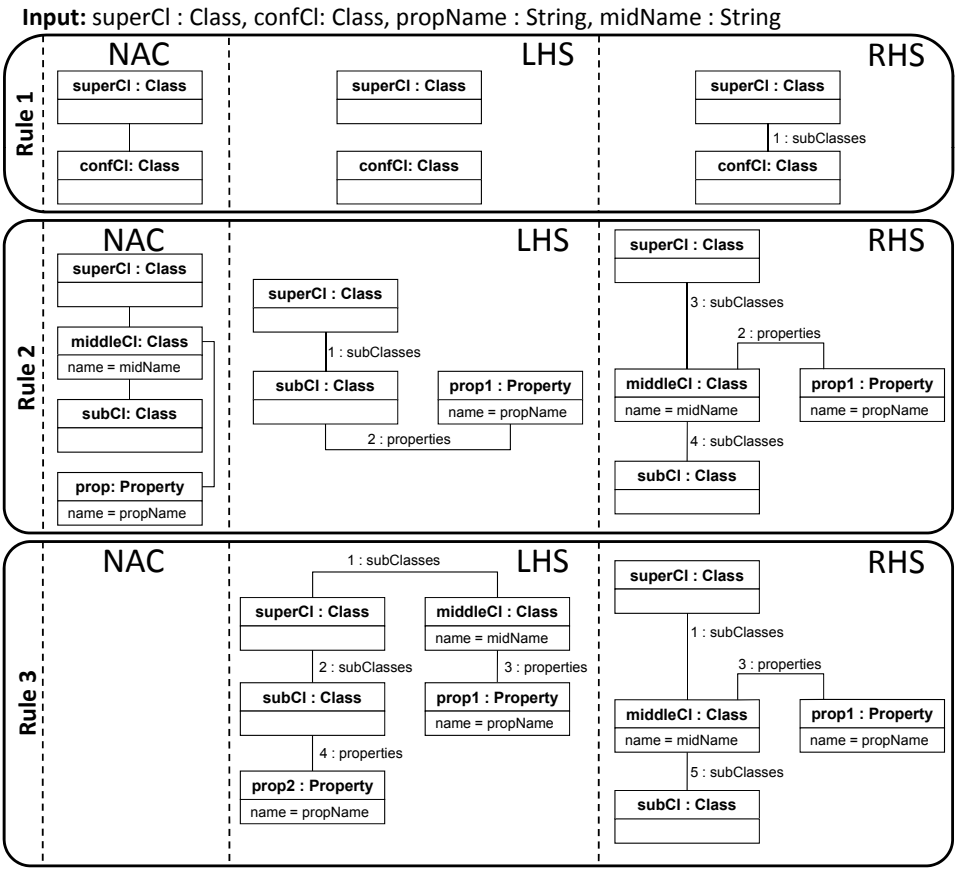

Fig. 4: Derived Resolution Pattern.

Concerning the example conflict, a graph transformation consisting of three rules is derived, which is illustrated in Figure 4. This pattern takes the superclass (cf. superCl), the class which violates the precondition of the refactoring (cf. confCl), and the name of the field to be shifted (cf. propName) as input. Moreover, the user has to provide the name of the newly introduced intermediate class (cf. midName). As conflicting operations are not automatically merged, the tentative merge neither includes the inheritance relationship between the conflicting class and the superclass nor the refactoring "Pull Up Field". Hence, both operations have to be captured by the resolution pattern. Rule 1 matches only once and adds the inheritance relationship, i.e. the pending operation of Sally. Rule 2 and Rule 3 are used to reflect the refactoring of Harry. Rule 2 introduces the intermediate class with the given name and links the property to be pulled up therewith. Due to the negative application condition matching the produced structure, this rule is either executed once. As long as subclasses comprising the given property exist, the property is removed from these subclasses by Rule 3. This pattern is stored in the *Resolution Pattern Storage* for reusing it in future scenarios.

Merge. The *Conflict Report* is used to construct the consolidated merged version. If the user has decided to apply suggested conflict resolution patterns, then they are automatically executed. Finally, the merged version is stored in the repository.

4 Derivation of New Resolution Patterns

A few days later, Sally gets in trouble again during the check-in. In order to put her new model version into the repository, she has to pass again the AMOR check-in process. As in the previous example Harry again has been faster with applying his changes and has already updated the head version of the repository. Hence, Sally's working copy has to be merged. Also this time she has a conflict with Harry's work. Fortunately, AMOR is now able to recommend a resolution strategy based on the previous example found in the *Resolution Pattern Storage*. In the following, we present a new versioning example and show how AMOR is able to automatically adapt and apply the afore derived resolution pattern for the new conflict.

4.1 Motivating Example 2.0

Again, Harry and Sally work on the common base model V0 depicted in Figure 5(a). The model contains an inheritance hierarchy of birds including the superclass Bird and the two subclasses Hawk and Duck. This time, both of the subclasses own an operation named getFlightSpeed(). In order to optimize the model's structure, Harry applies the refactoring "Pull Up Method" [FBB+99] to shift the common operation getFlightSpeed() to the superclass. In the meantime, Sally expands the model by inserting further subclasses Robin and Penguin. While the class Robin also has the operation getFlightSpeed(), the class Penguin has not. This is due to the fact that penguins cannot fly.

4.2 Metamodel-based Resolution Lookup

When checking-in her new version, AMOR supports Sally as follows.

1. *Conflict Detection.* When Sally checks in her working copy, the merge cannot be performed automatically. A conflict is reported because Harry has applied the refactoring "Pull Up Method" in parallel to Sally's modifications. In Sally's version, the preconditions for the application of the refactoring are not satisfied due to the absence of the operation which should be shifted to the superclass in the class Penguin. A *Conflict Report* is generated (cf. 5(b)).
2. *Resolution Lookup.* In this step, it is checked, whether suitable resolution patterns have already been defined. Therefore, the *Resolution Pattern Storage* is searched for candidate resolution patterns. If a perfect match is found in the *Resolution Pattern Repository*, only the associated resolution pattern has to be applied for resolving the conflict. However, if only a partial match is found, special reasoning

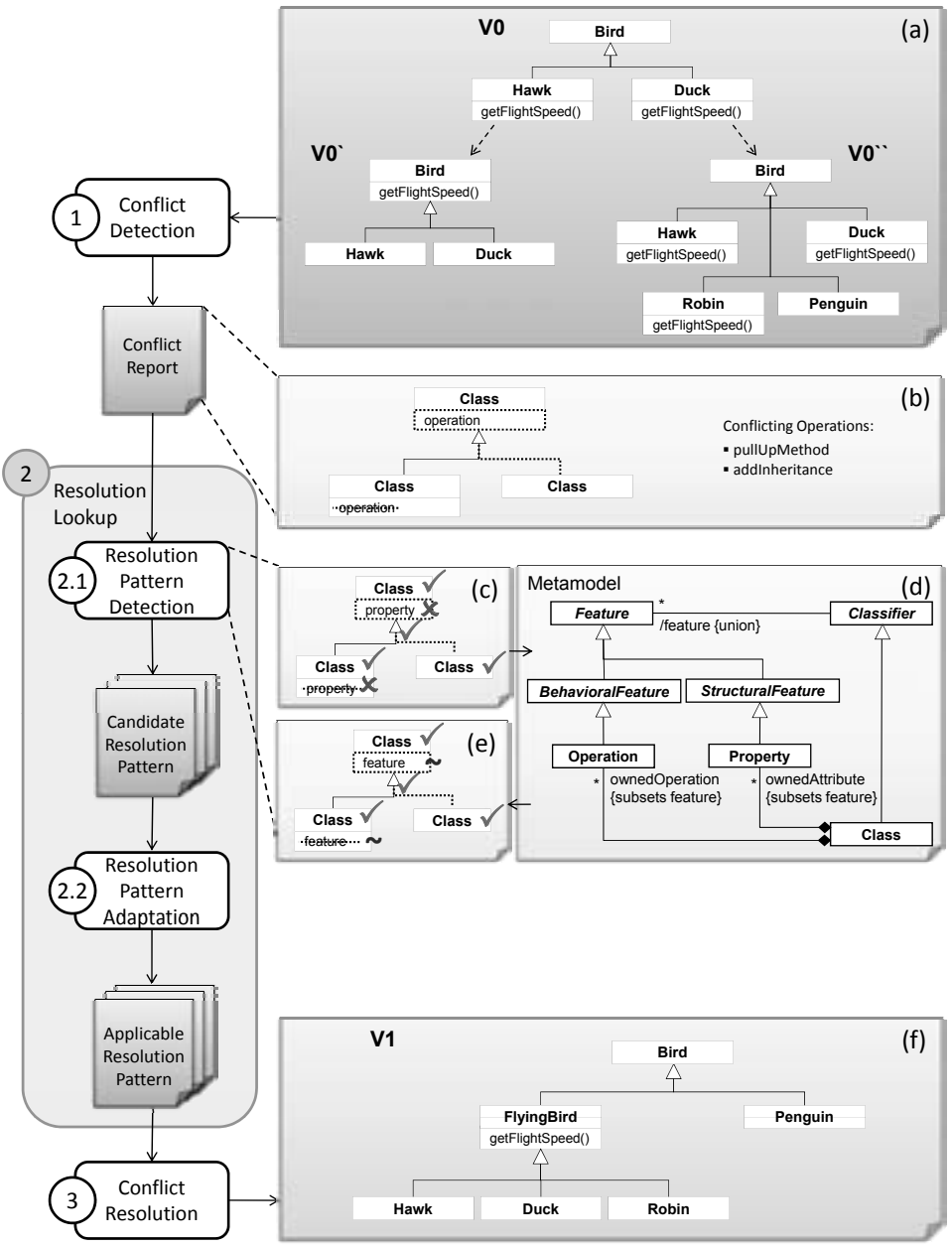

Fig. 5: Metamodel-based Resolution Lookup By-Example.

techniques have to be applied in order to decide if the partial matching conflict is generalizable to match the reported conflict. The generalization of partial matching conflicts requires an adaptation of the associated resolution pattern before it is ap-

plicable on the reported conflict which is done in the second phase called *Resolution Pattern Adaptation*.

2.1. *Resolution Pattern Detection.* With the *Conflict Report* at hand, the first phase in semi-automatic conflict resolution is the lookup of suitable conflict descriptions. A query, taking the *Conflict Report* as parameter, is sent to the *Resolution Pattern Storage* which not only returns resolution patterns for the actual exact conflict. It also returns resolution patterns which had been applied to similar scenarios and which might be adaptable to the actual one. This is done by first comparing the untyped graph structure of the model of the current conflict report with models already persisted in the *Resolution Pattern Storage*. Once matching graph structures are found, the type information is analyzed. For this, the metamodel is used as knowledge base for finding similarities of model elements.
For the new conflict situation, the conflict of the previous example (cf. Figure 1) is found as a similar conflict. The lookup algorithm detects the similar structure of the conflicting part of the model. By comparing the type information, a positive match is found for the classes and the inheritance relationship (cf. Figure 5(c)). However, instead of operations the classes contain properties. Instead of rejecting the found pattern immediately, a similarity analysis of the mismatching elements is performed. With the help of the metamodel (cf. Figure 5(d)) inheritance relationships between those elements are identified. By reflecting the metamodel, the algorithm finds out that Property and Operation both are subclasses of the Feature class and are both associated to Class which is relevant for both refactorings "Pull Up Field" and "Pull Up Method". Furthermore, the associations ownedAttribute and ownedOperation are subsets of feature. Hence, they may be considered as similar (cf. Figure 5(e)).
This analysis is accomplished for all matching graph structures found in the *Resolution Pattern Storage*. The most similar conflicts are returned ordered by similarity metrics and statistics of their previous usage.

2.2. *Resolution Pattern Adaptation.* For each candidate resolution pattern found in the previous step, a higher-order transformation [TJF+09] is applied in order to modify the corresponding resolution rule to fit the necessary model elements in the actual conflict situation. The higher-order transformation may replace the type information like in the example above. In our case, this means that the resolution pattern illustrated in Figure 4 is rewritten as followes. First, the types are substituted. Second, the links are adapted to match the new types. Third, links and attributes which are only existing for Property and not for Operation are eliminated. However, the last step is not necessary for our example.

3. *Conflict Resolution.* Regardless whether the user decides for a single or a collaborative conflict resolution, a tentative merge and a list of pending operations, which could not be applied so far, are presented to the user. In cases where similar conflicts were found in the *Resolution Pattern Storage*, the list of the adapted resolution patterns is also presented to the user. The user may now select one of the resolution patterns which is then automatically applied on the actual conflict situation. A preview of the resolution is displayed and manual adjustments are possible. In our example, the adapted resolution pattern is able to produce a new version which includes the

intentions of both modelers in a consistent way (cf. Figure 5(f)). Sally only has to provide a name for the newly introduced middle class.

According to step 4 of the AMOR workflow (cf. Figure 3), the new resolution pattern is derived by the *Resolution Reasoner* and persisted in the *Resolution Pattern Storage* in order to be recommended in future conflict situations. Finally in the Merge, the conflict is resolved and committed to the repository.

5 Related Work

In the following, we give a short overview of work related to the AMOR model versioning system. Therefore, we consider the three issues (1) model versioning in general, (2) refactoring-aware versioning, and (3) conflict resolution support.

Model 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 [CW98] and [Men02]. 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 [MCPW08] supports the versioning of UML models. This system performs the conflict detection at a very fine-grained level, hence it is able to merge modifications concerning different model elements or even different attributes of one model element. Odyssey-VCS neither considers composite operations nor does it provide user support in conflict situations. EMF Compare [BP08] is an Eclipse plug-in which is able to match, to compare, and to merge Ecore-based models. In combination with a model repository a model versioning system could be realized. EMF Compare is intended to match any kind of model artifact, hence no language-specific operations and conflicts can be handled. CoObRA [SZN04] is integrated in the Fujaba tool suite and logs the changes performed on an artifact, hence it is tightly-coupled to a modeling tool. 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 [Kög08], an Eclipse-based CASE-tool, integrates different model viewpoints. The Unicase client allows viewing and editing models in a textual, tabular, and diagramming visualization. The models are stored in a repository and can be versioned. The provided three-way merge technique makes use of editing operations, which are obtained from the Unicase client. SMoVer [Alt08] is a VCS for EMF-based model artifacts which is able to detect semantic conflicts. Therefore the parallel evolved model versions are transformed to a semantic view which emphasizes a certain semantic aspect. For each modeling language and each semantic view the transformation has to be manually specified.

Overall, these approaches focus on specific aspects of conflict detection and provide little or no resolution support at all. However, some of these approaches are suitable for the integration in AMOR. A detailed survey on model versioning systems is given in [ASW09].

Refactoring-Aware Versioning Systems. Our approach comprises knowledge on applied refactorings to improve the quality of the merge. Hence, it is related to refactoring-aware versioning systems such as [EA04] and [DMJN08]. Like in these approaches, we replay the applied refactorings after all atomic changes are merged. The check of the refactoring's preconditions before its execution in order to proof its applicability and detect potential refactoring conflicts is also done in [EA04]. In contrast to [EA04] and [DMJN08] we discuss refactoring-aware versioning for models and not code. Beside that, our approach has three main advantages over [EA04] and [DMJN08]. First, we do not restrict the language of the artifacts under version control. In AMOR any language based on Ecore is supported. Second, AMOR allows the user to define custom refactorings by-example and does not only support a fixed range of refactorings. Finally, AMOR does not rely on an editor-specific operation tracking which is done in [EA04] and [DMJN08]. Our approach detects refactorings state-based, i.e., only by analyzing the modified models and their common base version. Consequently, any editor may be applied to change and refactor the models.

Conflict Resolution Support. One of the key challenges in versioning is the resolution of conflicts in an automatic way [Men02]. To the best of our knowledge, automatic conflict resolution has not been achieved so far. As the conflict resolution involves many user decisions in general, total automatic resolution will probably never be possible. Semi-automatic approaches offering resolution suggestions to the user are realized for example in MolhadoRef [DMJN08] for code versioning. PROMPT [NM00], a tool for ontology versioning, uses a manual approach, but supports the user by pointing to conflicts as well as giving some hints how to resolve them.

AMOR provides a recommender component suggesting conflict resolution patterns, which are automatically executed, if the user has selected them. The resolution patterns are learned from conflict resolutions manually done by the users. However, therefore we need solutions of high quality capturing all user intentions. AMOR provides a synchronous and collaborative resolution approach (cf. [BLS+09a]) inspired by the work of Dewan and Hegde [DH07], who consider collaborative merge techniques only for software code but not for models.

6 Conclusion and Future Work

In this paper, we presented the adaptable model versioning system AMOR. AMOR extends a standard optimistic versioning system with respect to three aspects. (1) AMOR provides a conflict detection component which may be enhanced with user-defined composite operations. This additional information allows a more precise conflict detection on the one hand, and a more compact conflict report on the other. (2) AMOR provides a recommender component which offers suggestions how to resolve a previously detected conflict. These suggestions may be either manually defined or they are learned from the situations when the modelers resolve the conflicts by hand. (3) Finally, AMOR provides collaborative conflict resolution features, which allow the implementation of conflict resolution policies. In

order to ensure that the intentions of all modelers are captured in the merged version, it is sometimes necessary that they perform the conflict resolution together.

In future work, we will investigate the impact of different similarity heuristics for the lookup of similar conflicts in the *Resolution Pattern Storage*. These heuristics are of particular importance if legacy modeling tools are used which either have no metamodel or a less object-oriented one. Then it is not possible to make use of inheritance relationships. Also then the Lookup Component should be able to find applicable patterns. This will also involve more sophisticated higher-order transformations in order to adapt the transformations which incorporate the resolution solution in the merged model version.

Furthermore, we will perform an evaluation of the Conflict Detection Component of AMOR by defining multiple refactorings for the *Operation Repository* and running tests based on artificially provoked conflict scenarios. Finally, we will conduct an extensive case study in the context of a real world project.

References

[AKK+08] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. AMOR—Towards Adaptable Model Versioning. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management @ MoDELS'08*, 2008.

[Alt08] Kerstin Altmanninger. Models in Conflict — Towards a Semantically Enhanced Version Control System for Models. *Models in Software Engineering*, pages 293–304, 2008.

[AP05] Marcus Alanen and Ivan Porres. Version Control of Software Models. *Advances in UML and XML-Based Software Evolution*, 2005.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, 5(3), 2009.

[BLS+09a] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. We Can Work It Out: Collaborative Conflict Resolution in Model Versioning. In *Proceedings of the 11th European Conference on Computer Supported Cooperative Work, ECSCW'09*, pages 207–214, 2009.

[BLS+09b] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An Example is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems, MoDELS'09*, pages 271–285. Springer, 2009.

[BP08] C. Brun and A. Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 2008.

[CRP08] Antonio Cicchetti, Davide Ruscio, and Alfonso Pierantonio. Managing Model Conflicts in Distributed Development. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS'08*. Springer, 2008.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232, 1998.

[DH07] Prasun Dewan and Rajesh Hegde. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work, ECSCW'07*. Springer, 2007.

[DJ06] Danny Dig and Ralph Johnson. How Do APIs Evolve? A Story of Refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[DMJN08] Danny Dig, Kashif Manzoor, Ralph E. Johnson, and Tien N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[EA04] Torbjörn Ekman and Ulf Asklund. Refactoring-Aware Versioning in Eclipse. *Electronic Notes in Theoretical Computer Science*, 107:57–69, 2004.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[KGE09] Jochen Malte Küster, Christian Gerth, and Gregor Engels. Dependent and Conflicting Change Operations of Process Models. In *ECMDA-FA*, pages 158–173, 2009.

[Kög08] Maximilian Kögel. Towards Software Configuration Management for Unified Models. In *Proceedings of the 2nd International Workshop on Comparison and Versioning of Software Models @ ICSE'08*. ACM, 2008.

[MCPW08] Leonardo Murta, Chessman Corrêa, Joao Gustavo Prudêncio, and Cláudia Werner. Towards Odyssey-VCS 2: Improvements over a UML-based Version Control System. In *Proceedings of the 2nd International Workshop on Comparison and Versioning of Software Models @ ICSE'08*. ACM, 2008.

[Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[NM00] Natalya Fridman Noy and Mark A. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pages 450–455. AAAI Press / The MIT Press, 2000.

[OS06] Takafumi Oda and Motoshi Saeki. Meta-Modeling Based Version Control System for Software Diagrams. *IEICE Transactions on Information and Systems*, E89-D(4):1390–1402, 2006.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences Between Versions of UML Diagrams. *ACM SIGSOFT Software Engineering Notes*, 28(5):227–236, 2003.

[SZN04] Christian Schneider, Albert Zündorf, and Jörg Niere. CoObRA - A Small Step for Development Tools to Collaborative Environments. In *Proceedings of the Workshop on Directions in Software Engineering Environments @ ICSE'04*, 2004.

[TJF+09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *Proceedings of the 5th European Conference on Model Driven Architecture-Foundations and Applications*, pages 18—33. Springer, 2009.