A Recommender for Conflict Resolution Support in Optimistic Model Versioning [*]

Petra Brosch Martina Seidl Gerti Kappel

Business Informatics Group
Vienna University of Technology, Austria
{lastname}@big.tuwien.ac.at

Abstract

The usage of optimistic version control systems comes along with cumbersome and time-consuming conflict resolution in the case that the modifications of two developers are contradicting. For code as well as for any other artifact the resolution support moves hardly beyond the choices “keep mine”, “keep theirs”, “take all changes”, or “abandon all changes”.

To ease the conflict resolution in the context of model versioning, we propose a recommender system which suggests automatically executable resolution patterns to the developer responsible for the conflict resolution. The lookup algorithm is based on a similarity-aware graph matching approach incorporating information from the metamodel of the used modeling language. This allows not only the retrieval of recommendations exactly matching the given conflict situation, but also the identification of similar conflict situations whose resolution patterns are adaptable to the current conflict.

Categories and Subject Descriptors D.2.9 [*Software Engineering*]: Management—Programming teams

General Terms Design, Languages

1. Introduction

Contemporary software engineering is confronted with two major challenges: first, with the complexity of modern software systems and second, with the complexity of the software development process itself. To deal with the first challenge techniques of model-driven engineering (MDE) are employed which benefit from the abstraction power of models [1]. Instead of being artifacts applied for mere design and documentation purposes only, models are successfully leveraged as basis for compiling executable code. The second challenge is faced with adequate tool support enabling the effective management of the software development process to deal with the evolution of developed artifacts.

Very prominent representatives of such management tools are *version control systems* (VCS) supporting collaboration among the team members of a project [7]. *Optimistic version control systems* allow multiple, possibly globally distributed developers to modify the same artifact at the same time and independently of each other. If two modifications do not contradict each other then they may be easily merged into one new version of the artifact [11]. If two modifications are contradicting, a manual resolution has to be performed which is a repetitive, time-consuming, and error-prone task in general. The developer responsible for conflict resolution has to decide for either one of the alternatives or (s)he has to provide a completely new variant. Currently, promising approaches are developed for the conflict resolution in the context of code versioning [8, 9] but for model versioning, no tool support is provided.

Taking advantage of the models’ graph-based structure and their rich semantics, we propose a recommender system facilitating the conflict resolution in optimistic model versioning. In [3], we presented a categorization of typical conflicts in model versioning which allowed us to identify an initial set of reoccurring conflict situations and typical resolution patterns. Furthermore, we proposed an approach to automatically mine existing model repositories for the automatic identification of formerly applied conflict resolution patterns [2]. On this basis we obtain a collection of conflicts and executable resolution patterns yielding the knowledge base for the recommender system. Conflicts are represented as models, i.e., in a graph-based data structure. The lookup algorithm realizes not only exact graph matching, but also

[*] 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.

similarity-aware graph matching as it is done in the context of pattern mining in code repositories [17]. The goal of our approach is supporting modelers during the conflict resolution process in optimistic versioning even if they do not exactly know the necessary resolution pattern at the beginning.

This paper is outlined as follows. In Section 2 we shortly explain our motivation to enrich versioning systems with a recommender component. In Section 3 we introduce a model for conflicts. On this basis we are able to develop a measure for the similarity of conflicts in Section 4, which is applied and evaluated in the similarity-aware graph matching algorithm we use to identify suitable resolution patterns. In Section 5 we discuss the realization of the recommender system supporting conflict resolution in model versioning and conclude with a discussion of related research areas and future work in Section 6.

2. Motivation

When an optimistic versioning approach is followed, each developer works independently on his/her local copy and synchronizes his/her work from time to time with a central repository. As long as the modifications of the different developers do not interfere, i.e., as long as their modifications commute, there is almost no overhead by using the versioning system. The workflow, the two developers—let us call them Sally and Harry—have to pass through, is as follows. Sally and Harry 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. Instead of doing productive work, he is now occupied by integrating his modifications and the changes of Sally. The resolution of conflicts requires manual intervention because an automatic merge usually yields unsatisfactory results as in the example shown in Fig. 1. The model originally stored in the repository contains a UML Class Diagram consisting of the classes `PublicTransport`, `Subway`, and `Train`, whereas the latter two classes are subclasses of the first and each of them contains the attribute `railtrackWidth`. When Sally introduces a new class `Bus` into the hierarchy (V0′) and Harry performs the refactoring pullUpField which shifts the attribute `railtrackWidth` common to all subclasses into the superclass (V0″), a naive merge including all modifications would result in a model where a bus inherits the attribute `railtrackWidth` which probably does not reflect reality (V0′ + V0″ in Fig. 1).

In order to preserve the intentions of both developers, the conflict resolution of Harry should be as shown in V1 of Fig. 1. A new class `RailVehicle` should be introduced which is a subclass of `PublicTransport` and which inherits the `railtrackWidth` attribute to `Subway` and `Train`, but not to `Bus` which is nevertheless a subclass of `PublicTransport`. This resolution strategy is applicable whenever a conflict of a similar structure reoccurs no matter if the involved classes represent vehicles, creatures, or something else. Therefore, it would be extremely supportive if this pattern is suggested to the developer in charge of the resolution and if the pattern is automatically executed when it is selected by the developer.

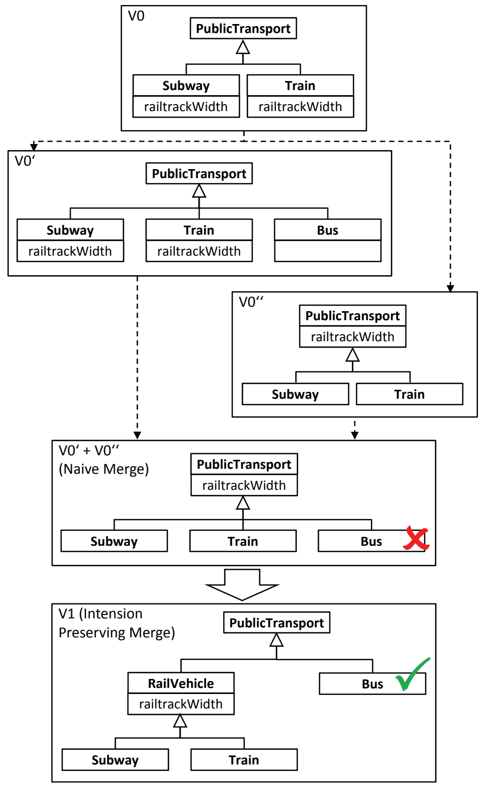


Figure 1. Conflict Scenario

Refactoring-aware versioning systems can detect and replay refactorings during the merge process to incorporate newly introduced and modified elements [8, 9]. When modifications violate the refactoring's precondition, a conflict is reported. Current versioning systems usually indicate only, where the modifications have taken place. Advanced conflict resolution support is not provided.

In the remainder of this paper, we present a recommender system as integral component of the adaptable model versioning system AMOR[1]. AMOR's sophisticated change and conflict detection component (the Conflict Detector) delivers precise information on merge problems [2]. Equipped with a repository filled with (conflict, resolution) pairs (the Conflict Repository), where the recommender system looks up suitable resolutions for a reported conflict, (semi-) automatic support for the conflict resolution in the context of model versioning is realized.

[1] `http://www.modelversioning.org`

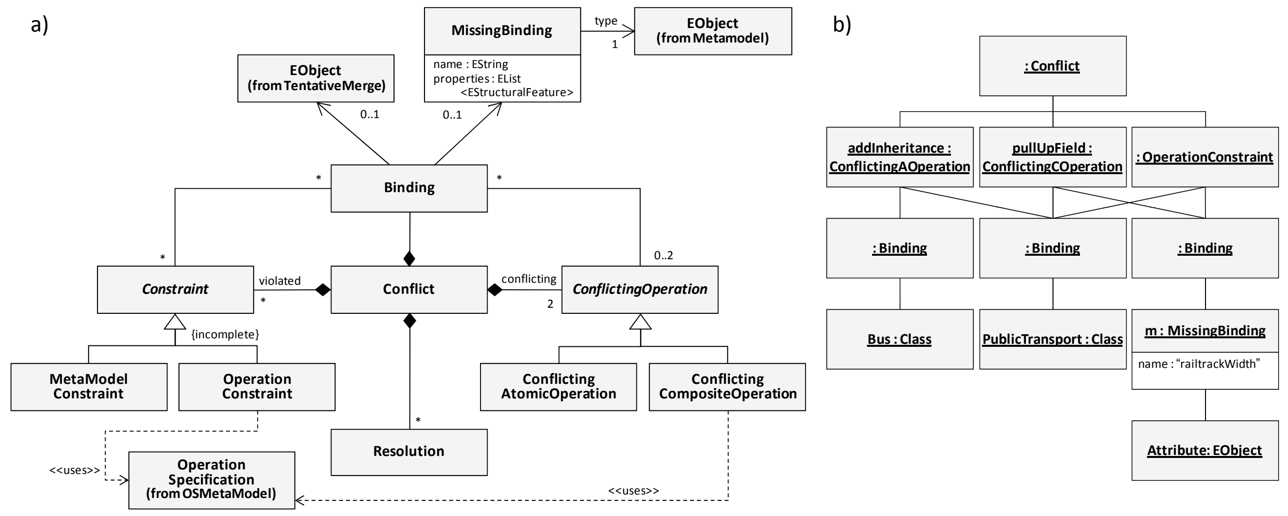


Figure 2. (a) Conflict Model, (b) Instance of the Conflict Model

3. Conflicts in Model Versioning

In the following, we represent conflicts by the means of a model, i.e., we define a UML Class Diagram for describing conflicts. This approach is similar to Cichetti et al. [6], where the authors present a conflict model as an extension to their difference model. Since the conflict detection component of AMOR [4] is able to reconstruct composite changes like refactorings, we obtain a more compact difference report and consequently a compact conflict description. Furthermore, we include status information of the merge process, i.e., we have information about the already integrated modifications given in the *tentative merge* model. The tentative merge consists of the origin version V0 merged with all non conflicting operations performed in the parallel edited versions V0' and V0'', hence the tentative merge model is a valid model incorporating the changes identified as unproblematic. Our conflict model is shown in Fig. 2(a). A `Conflict` always contains two conflicting operations. A `ConflictingOperation` is either a `ConflictingAtomicOperation` (e.g., *add*, *delete*, and *update*) or a `ConflictingCompositeOperation` (e.g., refactorings) which is based on AMOR's `OperationSpecification` as defined in [4]. The application of these two operations would induce the violation of some kind of `Constraint`. At the moment, we distinguish between two kinds of constraints: a `MetaModelConstraint` expresses a well-formedness rule of the applied modeling language. An `OperationConstraint` refers to an invariant, a precondition, or a postcondition of an operation. Note that a conflict does not contain all possible conditions and constraints but only the ones which are violated and hence important for the definition of the conflict. In fact, our conflict model provides a view on the elements of operation specifications where the elements necessary for the conflict description are included. The constraints and the operations of a conflict are related to specific bindings which represent their input arguments. These input arguments are expressed by either referring to an element of the tentative merge, or by specifying a `MissingBinding` if an element is not available in the tentative merge. A `MissingBinding` points to the type of the missing element in the underlying model and contains additional information about the missing element like its name and other properties. For a complete definition of our conflict model further well-formedness rules would be necessary. For example, it is necessary to ensure that the types of the bound elements of a `ConflictingOperation` are type compatible with the input parameters of the according `OperationSpecification`. If a binding is not assigned to an operation, it has to be assigned to at least one constraint. In this paper, we assume that AMOR's Conflict Detector provides syntactical correct conflict descriptions only, hence we omit these well-formedness rules. Finally, an arbitrary number of resolutions may be attached to each conflict.

An example of a conflict instance is shown in Fig. 2(b). The operations `pullUpField` and `addInheritance` are conflicting. The bindings point to the involved elements: the class `Bus` should become subclass of `PublicTransport`, whereas an attribute `Attribute` is moved to `PublicTransport`. Note that we have to deal with a missing binding, as the conflict exists due to the absence of an `Attribute` with the name `railtrackWidth`. For the application of the `pullUpField` operation, the following constraint (expressed in OCL) has to hold:

`PublicTransport.subclasses` → *forall* `(s |`
  `s.attributes` → *exists* `(a | a.name == m.name))`

The constraint is violated because there exists one subclass of `PublicTransport` (namely `Bus`) which does not have an attribute with the according name.

Conflicts as described in this example are returned from the Conflict Detector of AMOR. With such a conflict as input, the recommender system is able to look up suitable resolution strategies in the Conflict Repository using exact as well as similarity-aware matching techniques. When similarity-aware matching techniques are applied, three sources of variability may be considered: (1) the operations, (2) the conditions, and (3) the bindings. As a first step towards a similarity-aware conflict recognition, we consider the bindings in the following.

4. Finding Resolution Patterns

In the following example, we aim at illustrating the need for inexact matching techniques. Assume that the Conflict Repository contains only the conflict and its resolution presented in Section 2. The conflict scenario depicted in Fig. 3 emerges from the parallel modifications where one modeler introduces the new class `Penguin` into the inheritance hierarchy and the second modeler pulls up the operation `getFlightSpeed()` of the classes `Hawk` and `Duck` into the superclass `Bird`. A naive merge would produce a model where penguins are able to fly what contradicts reality. The Conflict Detector reports a conflict not due to this common domain knowledge, but due to a violated precondition of the refactoring pullUpMethod. When querying the Conflict Repository, no exact matching conflict is found. In order to find at least the conflict of the previous example (cf. Fig. 1), the ability to handle inexact matches is indespensable.

Unfortunately, existing matching tools (cf. [15] for a survey) or dedicated model diffing tools like EMF Compare are not appropriate for our purposes, since they operate on the model level only and do not consider similarities of the metamodel elements. SiDiff [14] implements a similarity-based algorithm which may be configured by the user. A configuration contains the impact of metamodel features. For example, the name of a class is an higher ranked similarity criterion than the value of the `isAbstract` feature. So the similarity of two model elements with the same type may be calculated using ranking information for the concrete instantiations. In contrast, we are also interested in the similarity of model elements with different types. Recall that information like the name of a model element is of little help for our purposes because we match the concrete conflict against a generic instance of the conflict model stored in the conflict repository.

Similarity of metamodel elements. One possibility to find a broader range of conflicts is to suspend type information and match on graph structure only. This approach may work well in many situations. The drawback is that structural equality of conflict model instances does not ensure the suitability of their resolution pattern.

A more reliable approach is to apply similarity-aware graph matching techniques [5]. Here the typed graph is analyzed, but inexactness is allowed as long as a minimum

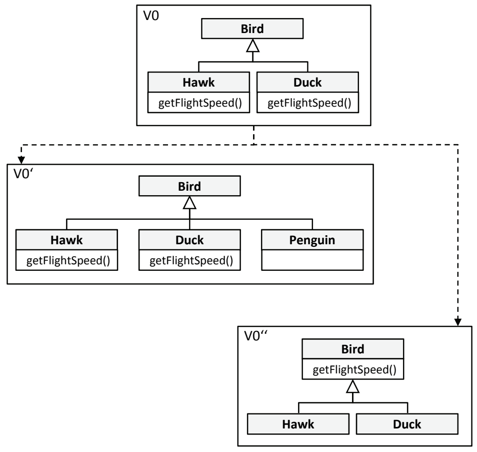


Figure 3. Conflict Scenario Revised

similarity of the compared nodes (i.e., model elements) is given. To faithfully support the generic, modeling language independent approach of AMOR, the necessary similarity measures should not be predefined, but be automatically derived by analyzing the metamodel of the modeling language in use. The taxonomic structure of the metamodel is an indication of the similarity between elements. Unfortunately, considering the inheritance alone is not enough. Some features like the name feature in a Class Diagram (cf. Fig. 4) are inherited to almost every element, hence it is a less valuable indicator for similarity. We propose to calculate similarity by exploiting the internal structure as well as the relational structure of metamodel elements. In fact, properties, relations, and inheritance relations are considered.

DEFINITION 4.1 (Similarity). *The similarity of two metamodel elements is given by the number of their common features weighted by their overall occurrence in the metamodel.*

For decreasing the weight of common features and contemporaneously increasing the weight of rare features, the frequency of the feature's appearance within the whole metamodel is considered like it is done by *term frequency-inverse document frequency* (TFIDF) algorithms [16]. TFIDF is usually used in information retrieval as a measure for the relevance of a term to a document. We use the metamodel as corpus and apply TFIDF as a measure for the relevance of a feature within a metamodel element.

Similarity algorithm. We implemented the algorithm within the Eclipse Modeling Framework (EMF) allowing the calculation of similarity values for every Ecore-based metamodel. A simplified variant of our approach is shown in Alg. 1. First, we analyze the metamodel and instantiate two lists. The list mmElements holds all elements defined in the metamodel, in line 1. The list features holds all distinct features occurring in any metamodel element (line 2). Second, in lines 3-5, we declare convenience functions for accessing

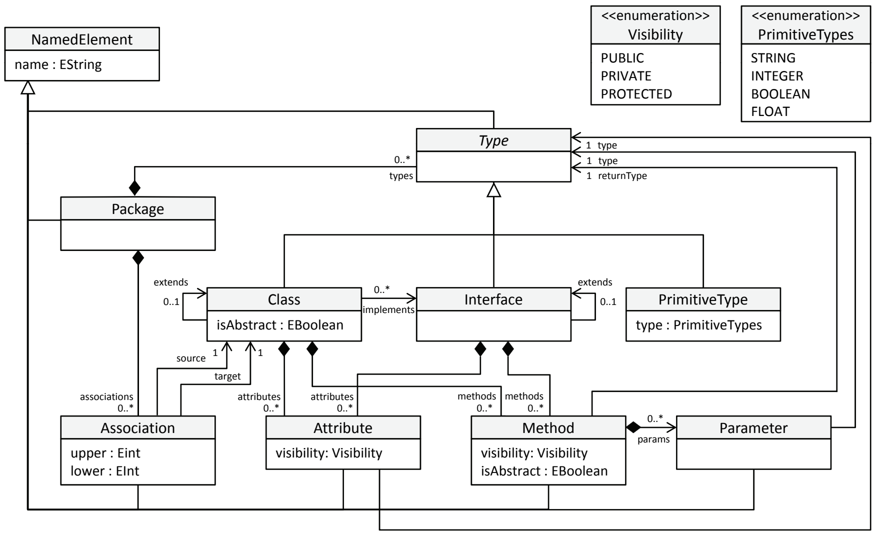


Figure 4. Class Diagram Metamodel

```
Input: Rootnode $root$ of metamodel
Output: Matrix of similarity values for each
        metamodel element pair
// variable declarations
1  mmElements $\leftarrow$ getAllMetamodelElements($root$);
2  features $\leftarrow$ getAllFeatures($root$);
// function declarations
3  relevanceMatrix : (EObject, Feature) $\mapsto$ float;
4  relevanceVector : (EObject) $\mapsto$ float$^{|\text{features}|}$;
5  similarityMatrix : (EObject, EObject) $\mapsto$ float;
// definition of relevanceMatrix
6  foreach MmElement $\mathcal{M} \in$ mmElements do
7      foreach Feature $\mathcal{F} \in$ features do
8          relevanceMatrix($\mathcal{M}, \mathcal{F}$) $\leftarrow$
           $\log\left(\frac{\texttt{countElements}(root)}{\texttt{getGlobalFreq}(\mathcal{F}, root)}\right) \cdot$
           getLocalFreq($\mathcal{F}, \mathcal{M}$);
9      end
10 end
// definition of similarityMatrix
11 foreach MmElement $\mathcal{N}, \mathcal{M} \in$ mmElements do
12     similarityMatrix($\mathcal{N}, \mathcal{M}$) $\leftarrow$
       $\|$relevanceVector($\mathcal{N}$)$\| \times \|$relevanceVector($\mathcal{M}$)$\|$;
13 end
14 return similarityMatrix
```

Algorithm 1: Similarity Calculation

arrays. Third, we calculate the relevanceMatrix in lines 6-10. The relevanceMatrix is a $n \times m$ array with values describing the relevance of each metamodel feature within each metamodel element. The number of metamodel elements is given by n, whereas m denotes the number of features occurring in the metamodel. The relevance of one feature within a metamodel element is based on the total number of metamodel elements, its occurrence frequency in the complete metamodel, and finally its occurrence frequency within the considered metamodel element. In the last step (lines 11-14), we calculate the similarity for each pair of metamodel elements by the cross product of the normalized relevance vectors. A relevance vector for one metamodel element $\mathcal{M}$ contains the relevance values of all features within $\mathcal{M}$, i.e., it respects to the line of the relevanceMatrix containing the relevance values of $\mathcal{M}$.

An example. In the following we apply the algorithm on a Class Diagram (cf. Fig. 4 for the metamodel). For the sake of readability, the metamodel follows a general design pattern but leaves out specific details. All elements extend directly or indirectly the common superclass `NamedElement`, enumeration types excluded. `Package` forms the root element and contains `Types` and `Associations`. `Class`, `Interface`, and `PrimitiveType` specialize `Type`. Both `Classes` and `Interfaces` may extend again a `Class` and an `Interface` and contain `Attributes` and `Methods`. In addition, a `Class` may be abstract and may implement `Interfaces`. `Attributes` and `Parameters` have a `Type`, a `Method` contains an arbitrary number of `Parameter` and returns a `Type`. In addition, well-formedness rules are necessary, e.g., neither a `Class` nor an `Interface` must extend itself.

The calculated similarity values for the metamodel elements of the Class Diagram shown in Fig. 4 are summarized in Table 1. Values greater than 0.1 are highlighted. As expected, a significant similarity between `Class` and `Interface` is found due to their number of common features. Furthermore, `Attribute`, `Method`, and `Parameter` are recognized as similar, because they all have the relation

	NamedElement	Package	Interface	Class	Association	Attribute	Method	Parameter	PrimitiveType	Type	Visibility	PrimitiveTypes
NamedElement	1,00	0,05	0,03	0,03	0,04	0,05	0,04	0,06	0,06	0,09	0,00	0,00
Package	0,05	1,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00
Interface	0,03	0,00	1,00	0,54	0,03	0,00	0,00	0,00	0,24	0,37	0,00	0,00
Class	0,03	0,00	0,54	1,00	0,02	0,00	0,09	0,00	0,20	0,31	0,00	0,00
Association	0,04	0,00	0,03	0,02	1,00	0,00	0,00	0,00	0,05	0,08	0,00	0,00
Attribute	0,05	0,00	0,00	0,00	0,00	1,00	0,54	0,29	0,00	0,00	0,00	0,00
Method	0,04	0,00	0,00	0,09	0,00	0,54	1,00	0,00	0,00	0,00	0,00	0,00
Parameter	0,06	0,00	0,00	0,00	0,00	0,29	0,00	1,00	0,00	0,01	0,00	0,00
PrimitiveType	0,06	0,00	0,24	0,20	0,05	0,00	0,00	0,00	1,00	0,65	0,00	0,00
Type	0,09	0,00	0,37	0,31	0,08	0,00	0,00	0,01	0,65	1,00	0,00	0,00
Visibility	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00	0,00
PrimitiveTypes	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,00	1,00

Table 1. Calculated Similarity Values for the Class Diagram Metamodel

to `Type`, and `Attributes` and `Methods` are both contained by the same elements. The inheritance relation of `Class`, `Interface`, and `PrimitiveType` with `Type` also leads to observable similarity. Since the relevance of features are first weighted by their overall occurrence in the metamodel and normalized within each metamodel element, each specific feature may have a different impact in different metamodel elements. The less features one element has, the higher the relevance of each feature is. Consequently, the similarity of subclasses of `NamedElement` to their superclass varies, but is nevertheless negligible because so many elements share this relationship.

Finally, we want to emphasize the general applicability of the presented approach. Although we developed the algorithm dedicated for matching conflict descriptions, it may be used for solving any kind of model matching problem if the metamodel of the modeling language is available.

With the components introduced in the previous section, we are able to realize a three-staged comparison algorithm which performs the following steps.

Exact match. The two typed graphs of the conflict models are exactly matched. Therefore, each node (resp. each edge) has to be matched against one node (resp. edge) with exactly the same type.

Match based on type compatibility. If the exact match fails, then the type restrictions are relaxed. Then elements of compatible types (i.e., sub- and superclass) are considered as equal.

Match based on type similarity. Finally, the elements having a similarity exceeding the required threshold are matched. The similarity is calculated once for the metamodel of each modeling language according to the algorithm presented above.

Implementation. The implementation of the presented conflict reasoning approach is mainly based on EMF and on the Java graph library JGraphT. The conflict model and the Class Diagram metamodel are implemented in Ecore and integrated as Eclipse Plug-ins into AMOR. AMOR's Conflict Detector turns over the conflict model, the tentative merge model, and the actual metamodel to the recommender system. The conflict model links to elements of the tentative merge for describing `ConcreteBindings` and to elements of the tentative merge's metamodel—in our case the Class Diagram metamodel—for defining the types of `MissingBindings`.

Before looking up for appropriate conflicts already included in the Conflict Repository, the similarity measures of metamodel elements have to be calculated. As long as the metamodel does not change, these values must be provided only once. The calculation is performed according to Alg. 1.

For comparing the conflict instances with each other in a similarity-aware manner, a graph isomorphism algorithm of the Java graph library JGraphT is used. Therefore, the conflict models are first converted to directed graphs. Model elements are represented by vertices and associations are represented by directed edges. To improve performance, the graph representations are hold in memory. The graph comparison algorithm may be either used to find exact matches, or to find equivalent matches. For finding equivalent matches, an implementation of the Interface `EquivalenceComparator` has to be provided. In our case, the implemented `ModelElementEquivalenceComparator` checks for equal types first, and in the case of a concrete binding or a missing binding's type, the pre-calculated similarity value of the type information is used to decide equality.

We are aware of the fact, that isomorph graph matching is assumed as NP-hard problem. As the number of conflicts stored in the Conflict Repository increases over time, the recommendation lookup gets slower. To avoid performance problems, we will use dedicated clustering techniques for graphs [10] to narrow the search space of conflict models in future work.

Experiments. We conducted first experiments to evaluate the applicability of our implementation in various conflict situations. Therefore, we selected a representative set of different conflict situations and matched them with our similarity-aware graph comparison algorithm. The operations involved in the conflicts are all applied on Class Diagrams as

		Conflicts in Repository					
		Cx	C3	C4	C5	C6	C7
Detected Conf.	C1	0					
	C2	0					
	C3		0	0.46	0.92		
	C4		0.46	0	0.46		
	C5		0.92	0.46	0		
	C6					0	0.46
	C7					0.46	0

C1	rename(c1:Class, "name1")
	rename(c1:Class, "name2")
C2	rename(a1:Attribute, "name1")
	rename(a1:Attribute, "name2")
C3	pullUpField(superCl:Class, a1:Attribute)
	addInheritance(subCl:Class, superCl:Class)
C4	pullUpMethod(superCl:Class, m1:Method)
	addInheritance(subCl:Class, superCl:Class)
C5	mvMethodToInt(m1:Method, i1:Interface)
	addInterfaceImpl(c1:Class, i1:Interface)
C6	addInheritance(c1:Class, c2:Class)
	addInheritance(c2:Class, c1:Class)
C7	addInheritance(i1:Interface, i2:Interface)
	addInheritance(i2:Interface, i1:Interface)

Table 2. Edit Distances of Conflict Scenarios.

defined by the previously introduced metamodel. The match is performed using the similarity matrix shown in Table 1.

Table 2 contains a short description of the detected conflicts. For more details we kindly refer to our project website. The conflicts for which resolutions are specified are arranged horizontally. In fact, we match each conflict against each conflict. The numbers in the table cells indicate the total edit distance between the conflict pairs. The edit distance is an indicator of the effort of rewriting the conflict resolution pattern, and is derived by summing up the edit distances of each vertex (1 - similarity). The conflicts C1 and C2 result from an update/update problem, as the name of the same element (a class in C1 and an attribute in C2) are concurrently modified in a different manner. Both of these conflicts are not included in the repository, but as no features of the specific classes are affected, the most general variant, in this case `NamedElement`, is stored which is denoted by Cx in Table 2. Note that for deducing a general conflict, also the features have to be considered which are involved in the conflict resolution. For matching the conflicts, considering the type compatibility is necessary, which results in an edit distance of 0 in Table 2. The conflicts C3 – C5 are variants of the previously presented motivating example, whereas C6 and C7 cause violations of the Class Diagram's metamodel by introducing inheritance cycles.

The empty fields in Table 2 indicate that no match has been found. All conflicts besides C1 and C2 may not be transformed to a more general form, hence we obtain exact matches in the diagonal (expressed by a 0). Summing up, in these first experiments, the algorithm shows the intended behavior. In future work, we will extend the scope of these experiments in the context of a broader case study.

5. Realization

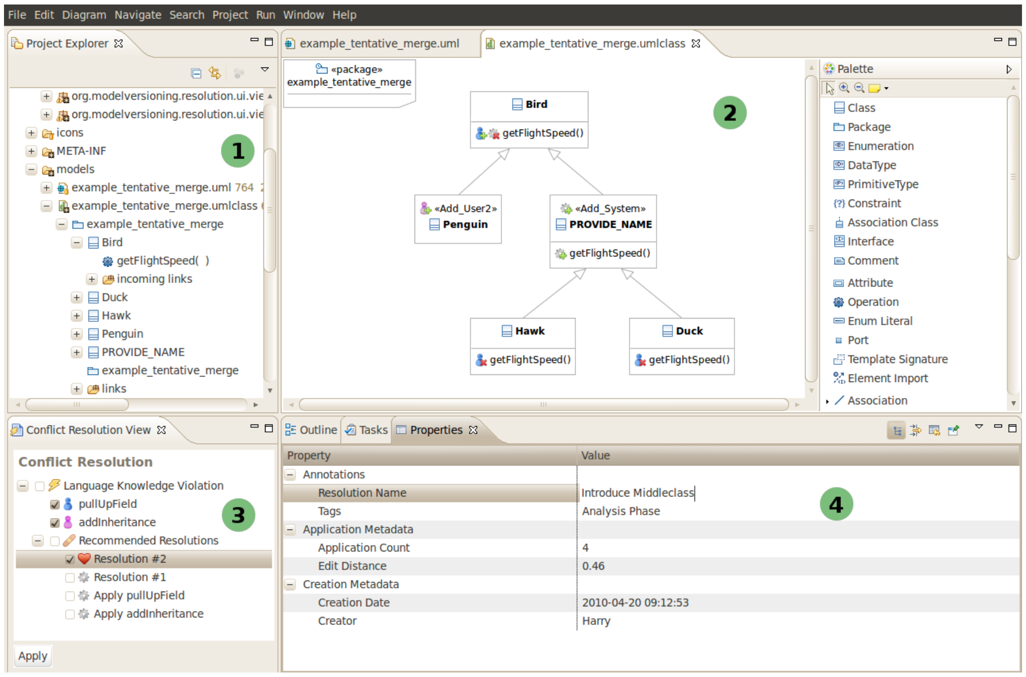
The recommender system is based on the Eclipse Modeling Framework the and Eclipse Team Support plugin. It implements the basic interplay with the versioning server (cf. (1) in Fig. 5) and offers the possibility to remodel artifacts to resolve conflicts (2). The actual recommender component (3) supports the conflict resolution by providing a list of automatically applicable resolution patterns for each conflict looked up from the Conflict Repository. The resolution patterns in the Conflict Repository are either defined manually or are automatically mined as described in [2]. The proposed resolution patterns may be previewed, rolled back, and manually refined. For easier identification of conflicting operations in the preview mode, the conflicting operations are marked with the dedicated user symbol combined with annotations indicating the application of *add*, *delete*, and *update* on the respective model elements. Recommended resolution strategies are marked accordingly with a system symbol (the cog). Previewing many operations at once may on the one hand overflow the model, but on the other hand may be necessary to understand changes. Therefore, the user may decide which operation should be displayed.

The recommended resolutions are ranked by their relevance. The relevance is calculated by a combination of the edit distance between the current conflict situation and the stored one, the number of applications so far, and the impact of the user who created the conflict resolution, by aggregating the application count of all resolutions created by this user. Resolution specific information is displayed in a dedicated property view (4). The property view contains metadata about the resolution's origin, application and edit distance. Furthermore, since automatically derived resolutions do not have a human understandable name, users may enhance the resolution pattern with additional information.

6. Conclusion

In the context of software engineering, recommender systems support developers in their decision making and particularly in their information finding goals [13]. Recommender systems for software engineering (RSSE) provide guidance for example in programming, i.e., by suggesting code for reuse, in debugging, i.e., by suggesting code for bugfixing, in testing, i.e., by indicating the parts of the program with the probably most defects, and in software maintenance. To support conflict resolution in versioning and especially in model versioning, to the best of our knowledge no recommender systems have been implemented yet. Current research focuses on the detection of differences and conflicts (like, e.g., SiDiff [14]) in order to support the resolution process without offering concrete resolution patterns. Only the ontology merging tool Prompt provides user guidance for a set of hard-coded conflicts [12].

In this paper, we introduced a recommender system enhancing the standard conflict resolution workflow of model



Conflict Resolution

- Language Knowledge Violation
  - [✓] pullUpField
  - [✓] addInheritance
- Recommended Resolutions
  - [✓] Resolution #2
  - [ ] Resolution #1
  - [ ] Apply pullUpField
  - [ ] Apply addInheritance

Property	Value
Annotations	
Resolution Name	Introduce Middleclass
Tags	Analysis Phase
Application Metadata	
Application Count	4
Edit Distance	0.46
Creation Metadata	
Creation Date	2010-04-20 09:12:53
Creator	Harry

Figure 5. AMOR Conflict Resolver

versioning by suggesting automatically executable resolution patterns. Conflicts are represented as Ecore models and are stored in a conflict repository. For the lookup of suitable resolution patterns we apply a novel kind of similarity-aware graph comparison algorithm allowing for exact matches, type compatibility matches, and type similarity matches.

In future work, we plan to conduct an extensive case study in cooperation with our industrial partner SparxSystems, the vendor of the modeling tool Enterprise Architect. In this context, we will also consider different modeling languages instead of the UML Class Diagram only. The gathered experiences will allow us to fine-tune our similarity calculation and to expand our conflict repository. Furthermore, much emphasis has to be spent on the user interface especially when the models get large. Then specific zooming and advanced filtering mechanism have to be implemented to avoid information overflow of the conflict resolving modeler.

References

[1] J. Bézivin. On the Unification Power of Models. *Journal on Software and Systems Modeling*, 4(2):171–188, 2005.

[2] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and H. Kargl. Adaptable Model Versioning in Action. In *Modellierung*, LNI. GI, 2010.

[3] P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *Int. Workshop on Model Comparison in Practice*. ACM, 2010.

[4] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Composite Operation Modeling By-Example. In *MODELS*, 2009.

[5] H. Bunke. Error-Tolerant Graph Matching: A Formal Framework and Algorithms. In *Advances in Pattern Rec.*, 1998.

[6] A. Cicchetti, D. Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *MODELS*, 2008.

[7] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Comp. Surv.*, 30(2), 1998.

[8] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Trans. on Software Eng.*, 34(3), 2008.

[9] T. Ekman and U. Asklund. Refactoring-Aware Versioning in Eclipse. *El. Notes in Theoret. Comp. Science*, 107, 2004.

[10] S. Günter and H. Bunke. Self-Organizing Map for Clustering in the Graph Domain. *Pattern Rec. Letters*, 23(4), 2002.

[11] T. Mens. A State-of-the-Art Survey on Software Merging. *IEEE Trans. on Software Eng.}, 28(5):449–462, 2002.*

[12] N. Noy and M. Musen. Algorithm and Tool for Automated Ontology Merging and Alignment. In AAAI*, 2000.*

[13] M. Robillard, R. Walker, and T. Zimmermann. Recommendation Systems for Software Eng. IEEE Software*, 2009.*

[14] M. Schmidt and T. Gloetzner. Constructing Difference Tools for Models using the SiDiff Framework. In ICSE Companion*. ACM, 2008.*

[15] P. Shvaiko and J. Euzenat. A Survey of Schema-Based Matching Approaches. Jnl. on Data Sem.*, 3730, 2005.*

[16] K. Spärck Jones. A Statistical Interpretation of Term Specificity and its Application in Retrieval. Jnl. of Doc.*, 28, 1972.*

[17] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. IEEE Trans. on Software Eng.*, 32(11), 2006.*